Exposing Disk Layout to Compiler for Reducing Energy Consumption of Parallel Disk Based Systems *

S. W. Son, G. Chen, M. Kandemir
CSE Department
Pennsylvania State University
University Park, PA 16802, USA
{sson,gchen,kandemir}@cse.psu.edu

A. Choudhary
ECE Department
Northwestern University
Evanston, IL 60208, USA
choudhar@ece.northwestern.edu

ABSTRACT

Disk subsystem is known to be a major contributor to overall power consumption of high-end parallel systems. Past research proposed several architectural level techniques to reduce disk power by taking advantage of idle periods experienced by disks. While such techniques have been known to be effective in certain cases, they share a common drawback: they operate in a reactive manner; i.e., they control disk power by observing past disk activity (e.g., idle and active periods) and estimating future ones. Consequently, they can miss opportunities for saving power and incur significant performance penalties, due to inaccuracies in predicting idle and active times. Motivated by this observation, this paper proposes and evaluates a compiler-driven approach to reducing disk power consumption of array-based scientific applications executing on parallel architectures. The proposed approach exposes disk layout information to the compiler, allowing it to derive disk access pattern, i.e., the order in which parallel disks are accessed. This paper demonstrates two uses of this information. First, we can do proactive disk power management, i.e., we can select the most appropriate power-saving strategy and disk preactivation strategy based on the compiler-predicted future idle and active periods of parallel disks. Second, we can restructure the application code to increase length of idle periods, which leads to better exploitation of available power-saving capabilities. We implemented both these approaches within an optimizing compiler and tested their effectiveness using a set of benchmark codes from the Spec2000 suite and a disk power simulator. Our results show that the compiler-driven disk power management is very promising. The experimental results also reveal that, while proactive disk power management is very effective, code restructuring for disk power achieves the best energy savings across all the benchmarks tested.

Categories and Subject Descriptors

B.4.3 [**Input/Output and Data Communications**]: Interconnections (Subsystems)—*Parallel I/O*; D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

General Terms

Algorithms, Design, Performance, Experimentation

Keywords

Optimizing Compiler, Parallel Disk, Low Power

1. INTRODUCTION AND MOTIVATION

Power consumption is becoming an increasing concern for high-performance parallel systems that execute large, data-intensive applications. There are several reasons for this. First, continuously increasing clock frequencies take power consumption to dramatic levels, as noted by several recent studies [12, 13]. Second, computing servers typically contribute to a large fraction of overall power budget of institutions and even cities [8, 6, 7]. Third, from an environmental viewpoint, reducing power consumption is desirable [1]. Therefore, several prior efforts considered hardware and software optimizations for reducing power consumption in high-end parallel systems.

Past research [14, 6, 8, 12] indicates that disk subsystems of parallel architectures can be a major power consumer. One way of reducing this power consumption is to adopt architectural mechanisms such as spinning down idle disks [10, 11, 18] or rotating disks with reduced speed [14, 6] when some amount of latency can be tolerated. A review of the prior work on disk power management is given in Section 2. While such techniques have been shown to be effective in certain cases, they have a common drawback: they operate in a *reactive* manner, that is, they control disk behavior based on observed disk activity (e.g., idle and active periods). In practice, this can bring two problems. First, they may fail to select the most appropriate disk power management scheme since their disk idleness estimation can be inaccurate. For example, if disk idleness is underestimated, these schemes behave conservatively in selecting the low-power mode to be employed. Consequently, they may not be able to use the most aggressive low-power mode. Second, they can incur performance penalties if they cannot determine accurately when an idle disk is going to be needed in the future. This is one of the most pressing problems facing parallel systems where disk requests coming from individual processors can interleave in time, and eventually make disk idle time (and active time) prediction very difficult.

Motivated by these observations, this paper proposes and evaluates a *compiler-directed* disk power management scheme targeting array-based scientific parallel applications executing on envi-

*This work was supported in part by NSF grants #0444158, #0406340, #0093082, and a grant from GSRC.

PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.

ronments with parallel disks. An optimizing compiler is in a very good position for the application domain and execution platform as stated above. This is because the compiler can analyze data access pattern of a scientific application based on a high level representation of the program [26], which enables users to capture how the disk resident data are accessed and shared by parallel processors. As for determining disk idle and active periods, extracting data access pattern alone may not be sufficient, and one actually needs the *disk access pattern*. We propose to obtain this pattern by *exposing* the layout information of disk resident data to the compiler. In other words, the proposed compiler support obtains disk access pattern by using data access pattern and disk layout information for array data. Section 3 explains the disk access pattern extraction process we proposed in detail.

After extracting the disk access pattern, this information can be used in at least two ways, both of which are explored in this study. First, one can implement a *proactive* disk power management strategy. What we mean by this is to let the compiler decide the times at which disks are switched to a low-power operating mode (e.g., spinning down a disk or operating it under reduced speed) and restored to the active status. As will be demonstrated in this paper, this proactive scheme can bring significant additional power benefits over the state-of-the-art hardware-based reactive strategies. Second, the compiler can restructure code to increase idle periods of disks, thereby allowing a more effective disk power management. We demonstrate that this restructuring can be expressed as a *scheduling problem*, which in turn can be handled by any known heuristic or exact scheduler. This paper discusses two variants of this scheduling problem, one that considers the problem from each processor's perspective independently and one that accounts for inter-processor disk sharing. Section 4 discusses proactive disk power management and Section 5 gives the details of our code restructuring strategy for reducing disk power, which is the main contribution of this paper.

We built a prototype of our approach using an optimizing compiler [15] and measured energy savings through a disk simulation environment. Our experimental results, obtained using several Spec 2000 benchmarks [25] with disk-resident data sets, show that, while proactive disk management is very effective, code restructuring achieves the best energy savings across all the applications tested. Our results also indicate that the benefits of our compiler-directed approach increases with increasing number of disks and data stripe sizes. Section 6 explains our experimental platform, simulation environment and benchmarks, and Section 7 presents experimental data. To test the behavior of our approach under different hardware and software parameters, we also conduct a sensitivity study in which we modify the default values of several simulation parameters used in our experimentation, and study their impact.

This study demonstrates that an optimizing compiler can be very successful in reducing disk energy consumption in a multiprocessor environment, provided that we can convey the disk layout information to the compiler thereby making the compiler aware of how data is striped (distributed) across parallel disks. Therefore, this paper discusses a different (non-traditional) usage of the compiler technology developed in the context of array-based parallel applications with regular data access patterns. The paper also shows that a compiler-directed scheme can be much more successful than the state-of-the-art hardware-based approaches to disk power management for array-intensive scientific applications.

2. DISCUSSION OF RELATED WORK

There has been significant past work on power management of high-end computing systems [7, 19, 9] and low-end embedded devices [21, 24, 3]. Due to space concerns, we limit ourselves in this section to disk energy optimization related studies. In particular, we focus mainly on two previously-proposed disk power management techniques.

The basic approach to save disk power is based on exploiting disk idle times, i.e., if there is enough idle time, the disk is spun down, meaning that it is transitioned into a low-power operating mode. The disk remains in the low-power mode until a new request arrives. This technique, denoted as TPM (traditional power management [10, 18, 11]) in this paper, has been extensively studied in the context of mobile disks since energy consumption in mobile systems is an important metric to minimize. Since a TPM disk operates in a reactive manner, i.e., the disk needs to be spun up before servicing a request, it incurs some performance penalty in general. To cut this potential performance penalty, determining a threshold value for idle period by employing either fixed or adaptive approaches is crucial in TPM. In this context, the threshold value is the minimum duration of idleness for which TPM makes sense. Although TPM is good mechanism for conserving disk power in laptop systems and embedded environments, recent studies also show that it is not a preferable option in the server or cluster domains, due to two reasons. First, the access patterns in server workloads are mainly small and non-contiguous, and consequently, disk idle times are not long enough to accommodate TPM. Second, for performance reasons, server class disks are operated at a very high RPM (revolutions per minute), typically above 10K RPM, and the disk spin-up/down times are really long, which in turn makes the threshold value very large.

Since exploiting idle time is hardly a viable option in the server class disks, [14] proposed dynamic RPM (referred to as DRPM in this paper) in which the disk hardware/controller provides several RPM steps. Note that, the higher RPM a disk spins at, the faster it services the I/O requests, and the higher power it consumes. An application that executes on a platform with DRPM capability can select disk speed dynamically at runtime to achieve the optimal balance point between energy consumption and execution time. In a sense, DRPM is similar in principle to CPU voltage scaling techniques proposed in literature because the selection of RPM step is made based on the change in the average disk response time recorded for n-request windows. Note that DRPM also incurs performance penalty because a lower RPM can potentially degrade response time. This can occur because a hardware-based DRPM strategy (like TPM) works with an estimation of disk idle times. If the estimation is not accurate, DRPM can select a wrong disk speed. It has been observed by the prior research [14] that DRPM can save significant amount of disk power by exploiting even small idle times, and it incurs relatively small performance penalty compared to TPM. Another technique based on modulating disk speed has been proposed and evaluated in [6]. In the rest of this paper, the term "low-power mode" (or "low-power state") refers to either a disk which is spun down (in TPM) or a disk whose speed is set to a lower RPM than the maximum RPM supported by the architecture (in DRPM). The focus of our approach is on maximizing the effectiveness of TPM and DRPM by scheduling the order of disk accesses in parallel disk based systems. Therefore, our approach can work with both TPM and DRPM based I/O systems.

In [16], Heath et al. describe an application code transformation technique for energy/performance-aware device management. Our work is different from their work in three aspects. First, we exclusively focus on disk power management by making use of proactive power mode selection and by employing a code transformation strategy oriented towards increasing disk idle times. In comparison, [16] tries to make best use of available buffer space. Consequently,

the code transformation approaches employed by the two studies are entirely different from each other (as the objectives are different). The second difference is that our focus is on server/cluster disks, while [16] concentrates on laptop disks. Therefore, they do not address the problem of compiler-driven use of DRPM, which is the main mechanism to save power in server class disks. Third, since these two efforts target different execution environments, they use different set of applications. In contrast to [16], our focus is on array-intensive scientific applications that spend a large fraction of their power budget on the disk subsystem. Since the strategy proposed in [16] is a generic scheme (not exclusively for disks), one can also envision it to co-exist with our scheme under a unified optimization framework.

3. DISK ACCESS PATTERN EXTRACTION

Our focus is on array based scientific applications with affine references. One of the important characteristics of these applications is that their data access patterns can be analyzed by an optimizing compiler and can be reshaped for different purposes such as optimizing data locality or improving parallelism.

One of the requirements for being able to use a compiler in reducing disk power consumption is to capture how parallel disks are accessed at a high level (i.e., source code level). We use the term *disk access pattern* in this paper to refer to the high-level information on the order in which parallel disks are accessed by a given application code. This order is important since it determines, for each disk in the system, active and idle periods, which is the primary information used for power management as explained in Section 2. Disk access patterns can be extracted at the loop iteration, loop nest, procedure, or even larger granularities. To obtain this information, the compiler needs data access pattern of the application code being optimized and disk layout information for array data (see Figure 1(a)). The first of these can be obtained by analyzing the application source code. Since such an analysis is performed by many optimizing compilers for different purposes (e.g., optimizing loop-level parallelism or cache locality), we do not discuss its details in this paper. As for the second parameter needed, we propose to *expose* the disk layout information to the compiler. In this way, the compiler will be aware of how array data is striped across the parallel disks, and can optimize the code accordingly.

We next discuss what type of disk layout abstraction is needed by the compiler in the proposed approach. File striping is a technique that divides a large data into small portions and stores these portions on separate disks in a round-robin fashion (as depicted in Figure 1(b)). This permits multiple processes to access different portions of the data concurrently without much disk contention. While striping can be performed manually, many file systems today provide automatic support for it, as will be explained below. In this work, we represent disk layout of an array using a triplet of the form:

(starting_disk, stripe_factor, stripe_size).

The first component in this triplet indicates the disk from which the array is started to get striped. The second component gives the number of disks used to stripe the data, and the third component gives the stripe (unit) size. Note that the several current file systems and I/O libraries for high-performance computing provide APIs to convey them the disk layout information when the file is created. For example, in PVFS [23], one can change the default striping parameters by setting `base` (the first I/O node to be used), `pcount` (stripe factor), and `ssize` (stripe size) fields of the `pvfs_filestat` structure. Then, the striping information

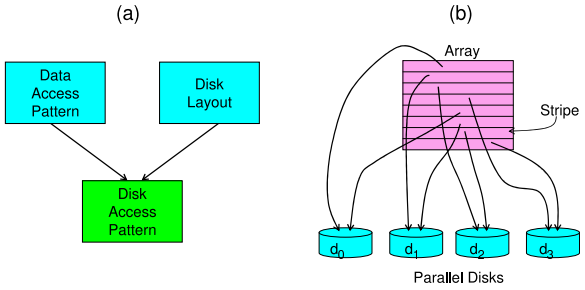


Figure 1: (a) Determining disk access pattern. (b) Striping an array over four disks.

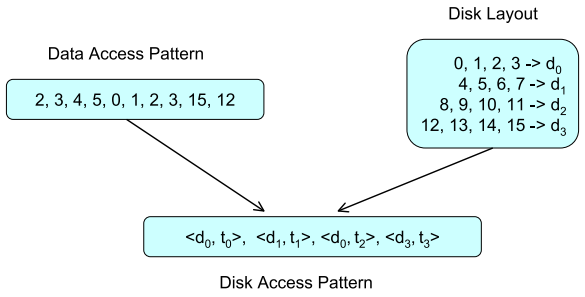


Figure 2: A data access pattern and the corresponding disk access pattern. $< d_i, t_j >$ means that disk d_i is (estimated to be) used for t_j cycles.

defined by the user via this `pvfs_filestat` structure is passed to the `pvfs_open()` call's parameter. When creating a file from within the application, this layout information can be made available to the compiler as well, and, as explained above, the compiler uses this information in conjunction with the data access pattern it extracts to determine the disk access pattern. On the other hand, if the file is already created on the disk system, the layout information can be passed to the compiler as a command line parameter.

The important point to note here is that we assume each data array manipulated by the application is stored in a separate file in the I/O system[1]. Since each file can have a different triplet of the kind shown above, each array can have a different disk layout than the others. While determining power-efficient disk layouts itself is an interesting research topic that we want to tackle in the future, in this paper we concentrate on code restructuring for low power. As a consequence, we assume that the disk layout information is given to the compiler, which subsequently uses it for determining disk access patterns.

Figure 2 shows a sample data access pattern and the corresponding disk access pattern. This disk access pattern is obtained under the disk layout shown in the same figure. In this layout, for illustrative purposes, the twelve elements of an array are distributed (striped) across four disks (d_0 through d_3). In the disk access pattern, a $< d_i, t_j >$ means that disk d_i is used for t_j cycles. t_j is estimated by the compiler. It is to be noted that the compiler can represent a disk access pattern using different representations and with different granularities. Since a given disk access pattern captures idle and active periods for each disk and their durations, it can be used for proactive power management (Section 4) or to restructure code to increase idle periods (Section 5).

[1]Our approach can be modified to handle other scenarios as well, e.g., multiple arrays per file, or multiple files per array.

4. PROACTIVE DISK POWER MANAGEMENT

After extracting disk access patterns, the compiler can insert explicit disk power management calls (instructions) in appropriate places in the source code. The purpose of these calls varies based on the underlying disk capabilities (e.g., TPM versus DRPM). For TPM disks, we use `spin_up()` and `spin_down()` calls. The format of the `spin_down()` call is as follows:

$$\texttt{spin\_down}(d_i),$$

where $\texttt{disk}_i$ is the disk id. Since a disk access pattern indicates not only idle times but also active times anticipated in the future, we can use this information to *preactivate* disks that have been spun down by a `spin_down()` call. To determine the appropriate point in the code to start spinning up the disk (that is, preactivation point), we take accounts of the spin-up time (delay) of the disk (i.e., the time it takes for the disk to reach its full speed where it can perform read/write activity). Specifically, the number of loop iterations before which we need to insert the spin-up (preactivation) call can be calculated as:

$$Q_{su} = \lceil \frac{T_{su}}{s + T_m} \rceil,$$

where Q_{su} is the preactivation distance (in terms of loop iterations), T_{su} is the expected spin-up time, T_m is the overhead incurred by a `spin_up` call, and s is the number of cycles in the shortest path through the loop body. It is to be noted that T_{su} is typically much larger than s. The format of the call that is used to preactivate (spin up) a disk is as follows:

$$\texttt{spin\_up}(d_i),$$

where as before d_i is the disk id. Note that, if we do not use preactivation, a TPM disk is automatically spun up when an access (request) comes; but, in this case, we incur the associated spin-up delay fully. The purpose of the disk preactivation is to eliminate this performance penalty. While our discussion so far has focused on the TPM disks as the underlying mechanism to save power, this compiler-driven proactive strategy can also be used with DRPM disks. The necessary compiler analysis and the disk access pattern construction process in this case are the same as in the TPM case. The main difference is how the disk access pattern collected is used (by taking the times to change disk speed into account) and the calls inserted in the code. In this case, we employ the following call:

$$\texttt{set\_RPM}(\texttt{rpm\_level}_j, d_i),$$

where d_i is the disk id, and $\texttt{rpm\_level}_j$ is the j^{th} RPM level (i.e., disk speed) available. When executed, this call brings the disk in question to the speed specified. The selection of the appropriate disk speed is made as follows. Since the transition time from one RPM step (level) to another is proportional to the difference between the two RPM steps involved [14], we need to consider the detected idle time to determine the target RPM step. Consequently, we select an RPM level if and only if it is the slowest available RPM level that does not degrade the original performance.

It must be mentioned that a wrong placement of the `spin_up()`, `spin_down()`, and `set_RPM()` calls in the code does *not* create a correctness issue. In the worst case scenario, they increase execution cycles and/or energy consumption. For example, prematurely spinning down a disk (in the TPM-based architecture) delays the time to service the next request, and leads to some extra energy consumption. Similarly, selecting a wrong RPM level to use (in the DRPM-based architecture) can increase disk energy consumption (if the selected level is faster than the optimal one) or execution time (if the selected level is slower than the optimal one). In either case, however, this is not a correctness issue. Notice however that the compiler places these power management calls into the code based on the disk access pattern it constructs for each disk. Since the compiler is conservative in handling the control flow within the loop bodies (i.e., it assumes that all branches of a conditional construct can be taken at runtime with an equal probability), the information it extracts (regarding disk idle/active times) may not be hundred percent accurate. The experimental results presented in this paper include such inaccuracies arising from the imperfect knowledge of the future access patterns. Notice also that while this compiler-directed proactive management can be very effective in reducing disk power (as will be shown by our experimental analysis), one can go beyond this by restructuring the source code so that disk reuse can be increased significantly. The second contribution of this paper is such a compiler-guided code restructuring strategy, and is explained in the next section in detail.

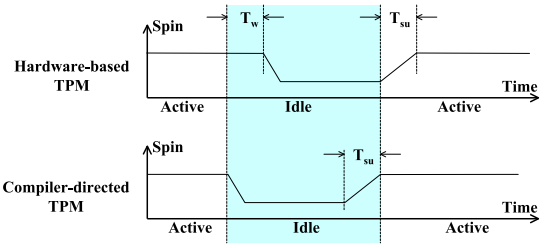


Figure 3: Comparison of the hardware-based TPM and the proposed compiler-directed TPM. In the hardware-based scheme, period T_w is for detecting idleness and T_{su} is the spin-up latency. The compiler-directed scheme can eliminate the impact of both these latencies.

Figure 3 illustrates the difference between the hardware-based TPM and the compiler-directed TPM. Compared to the hardware-based TPM, our approach has two advantages. First, the compiler-directed TPM can put idle disks in low-power mode earlier than the hardware-based TPM can do. Second, the compiler-directed TPM can avoid the performance overhead, using preactivation, due to the spin up latency when an idle disk is accessed. Figure 4 presents our compiler algorithm for disk energy optimization.[2] Our algorithm works in two steps. In the first step, we build a *Loop Transition Graph* (LTG) for a given procedure.[3] Each node L_i in the LTG corresponds to a loop nest in the procedure. A loop nest whose execution time is longer than a given threshold Q is recursively broken down into smaller loop nests until no loop nest contains any internal loop, or the execution time of the loop is shorter than Q. Each edge (from L_i to L_j) in LTG has a tag $C_{i,j}$, indicating the condition under which the flow of execution transitions from loop nest L_i to L_j. Figure 5(b) shows an LTG for the code fragment in Figure 5(a). In the second step, our algorithm inserts code to the program to spin up/down the disks. Specifically, for each node L_i in LTG, our algorithm inserts, before the entry of L_i, the `spin_down` calls for the disks that are not accessed in L_i. Further, if node L_i has a successor L_j that accesses a disk that has been spun down in L_i, we

[2]Due to lack of space, we give the formal algorithm only for inserting `spin_up` and `spin_down` calls. The algorithm for inserting the `set_RPM` calls is similar.

[3]Our current implementation is applied to each procedure separately; i.e., we do not perform any inter-procedural optimization.

```
procedure loopTransformation() {
  buildLTG();
  transform();
}

procedure buildLTG() {
  for each outermost loop L_i
    addNode(L_i);
  for each node (L_i) in the LTG
    determine disk access pattern D_i;
  for each pair of nodes (L_i and L_j) in the LTG
    determine transition condition C_{i,j};
}

procedure addNode(L_i) {
  if(execTime(L_i) > Q and L_i contains inner loops) {
    for each outermost loop L_j in L_i
      addNode(L_j);
  } else {
    add node L_i to the LTG;
  }
}

procedure transform() {
  for each node L_i in the LTG {
    if(execTime(L_i) > Q) {
      for each disk d_x
        if(D_i[x] = 0)
          insert before the entry of loop nest L_i:
            "spin_down(d_x)";
        if(exists L_i --C_{i,j}--> L_j such that d[j]&d[i] ≠ d[j]) {
          split L_i into two consecutive loop nests: L'_i and L''_i
            such that execTime(L''_i) = Q_su;
          for each disk d_x such that D_i[x] = 0
            for each loop nest L_j such that L_i --C_{i,j}--> L_j
              if(D_j[x] = 1)
                insert before the entry of of L''_i:
                  "if(C_{i,j}) spin_up(d_x)";
    }
  }
}
```

Figure 4: Compiler algorithm for inserting disk power management calls in a given code fragment.

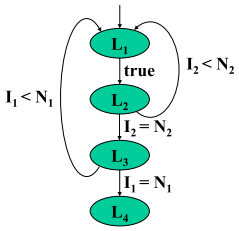
split L_i into two consecutive loop nests, L'_i and L''_i, such that the execution time of L''_i is equal to Q_{su}, the time required to spin up a disk. Before L''_i, our algorithm inserts the `spin_up` calls for the disks that will be used in L_j. By doing this transformation, we hide the performance overhead due to disk spin up. That is, as explained earlier, this preactivation eliminates potential performance penalty. Figure 5(a) shows an example code fragment, and Figure 5(b) gives the corresponding LTG. Figure 5(c) is the transformed code fragment after applying our algorithm.

5. CODE RESTRUCTURING FOR REDUCING DISK ENERGY CONSUMPTION

In this section, we present a strategy that restructures a given procedure for increasing the benefits that could be obtained from the proactive scheme discussed above. This code restructuring approach operates on a graph representation called the *Inter-Processor Disk Access Graph* (or IDAG for short). An IDAG is composed of a number of *Processor Disk Access Graph*s (PDAGs). Each node in an IDAG represents a set of loop iterations (as will be explained shortly), and the directed edges between nodes capture data dependences.

```
for I_1 = 0 to N_1 {
  for I_2 = 0 to N_2 {
    L_1: for I_3 = 0 to N_3
      access: d_0, d_1;
    L_2: for I_4 = 0 to N_4
      access: d_0, d_3;
  }
  L_3: for I_5 = 0 to N_5
    access: d_4;
}
L_4: for I_6 = 0 to N_6
  access: d_3;
```

(a) Original code fragment.



(b) Loop Transition Graph (LTG) for the code fragment in (a). Nodes L_1, L_2, L_3, and L_4 correspond to the loop nests with labels L_1, L_2, L_3, and L_4, respectively.

```
for I_1 = 0 to N_1 {
  for I_2 = 0 to N_2 {
    spin_down(d_2, d_3);
    L'_1: for I_3 = 0 to N_3
      access: d_0, d_1;
    if(true) spin_up(d_3);
    L''_1: for I_3 = N_3 - Q_su to N_3
      access: d_0, d_1;
    L'_2: spin_down(d_1, d_4);
    for I_4 = 0 to N_4 - Q_su - 1;
      access: d_0, d_3;
    if(I_2 < N_2) spin_up(d_1);
    if(I_2 = N_2) spin_up(d_4);
    L''_2: for I_4 = N_4 - Q_su to N_4
      access: d_0, d_3;
  }
  spin_down(d_1, d_2, d_3);
  L'_3: for I_5 = 0 to N_5 - Q_su - 1
    access: d_4;
  if(I_1 < N_1) spin_up(d_0, d_1);
  if(I_1 = N_1) spin_up(d_3);
  L''_3: for I_5 = N_5 - Q_su to N_5
    access: d_4;
}
spin_down(d_0, d_1, d_2, d_4);
L_4: for I_6 = 0 to N_6
  access: d_3;
```

(c) Transformed code fragment. The loops L_1, L_2, and L_3 in (a) are split. For example, loop L_1 is split into L'_1 and L''_1, and the estimated execution time of L''_1 is equal to Q_{su}.

Figure 5: An example that illustrates proactive disk power management.

We assume that the set of loop iterations that will be executed by each processor has already been determined prior to approach. For this purpose, either user-assisted (e.g., [20]) or compiler-directed (e.g., [2]) code parallelization methods can be employed. The selection of the method to be used for assigning loop iterations to parallel processors in the system is orthogonal to the focus of this paper. Let $\mathcal{I}_p$ represent the set of iterations assigned to processor p (as a result of loop parallelization), where $0 \leq p \leq P-1$. We note that, for any legal parallelization scheme, we have:

$$\bigcup_{p=0}^{P-1} \mathcal{I}_p = \mathcal{I}_{total},$$

where $\mathcal{I}_{total}$ is the set of total iterations in the procedure (including all the loop nests).

We attach a *tag*, denoted T, consisting of D bits, where D is the number of parallel disks in the I/O system, to each iteration I in $\mathcal{I}_p$. A bit in the d^{th} position of T ($0 \leq d \leq D-1$) is 1 if and only if loop iteration I accesses disk d.[4] Otherwise, we set this bit to 0. For the sake of explanation, we assume existence of a function called $tag()$ that gives the tag of any iteration I, given as input. Now, we can classify the iterations in $\mathcal{I}_p$ into 2^D classes. The common characteristic of the iterations assigned to a class is that they have the same tag. In mathematical terms, we have:

$$\mathcal{I}_{p,T} = \{I \quad | \quad I \in \mathcal{I}_p \quad \wedge \quad tag(I) = T\},$$

that is, $\mathcal{I}_{p,T}$ holds the loop iterations that are assigned to processor p and have the tag T.

[4]Our approach is conservative in the sense that if I *may* access disk d (depending on conditional execution flow at runtime), we conservatively set the corresponding bit to 1.

Note that, from the disk power management perspective, it is beneficial to execute iterations in $\mathcal{I}_{p,T}$ one after another. This is because all the iterations in this set access the same set of disks, and the remaining disks can be placed into a low-power mode during these accesses to save power. However, it is also important to determine a good execution order for different $\mathcal{I}_{p,T}$s. In Sections 5.1 and 5.2, we present scheduling schemes, where the problem is considered from single processor's perspective and multi-processors' perspective, respectively. What we mean by "scheduling" in this context is an order in which the nodes in an IDAG (or PDAG when considering from the perspective of a single processor) are executed. In Sections 5.1 and 5.2, we explain our approach, assuming that PDAGs (or IDAG) in question are *cycle-free*. Later in Section 5.3, we discuss code transformations to eliminate cycles in IDAG/PDAGs. After these code restructurings, the resulting code is further modified by inserting the proactive disk power management calls as has been discussed in Section 4.

5.1 Single Processor Perspective

Each $\mathcal{I}_{p,T}$ class (set of iterations) is represented by a node in PDAG$_p$, the PDAG for processor p. We can formally define a data dependence from $\mathcal{I}_{p,T}$ to $\mathcal{I}_{p,T'}$ as follows:

$$dep(p,T,T') = \begin{cases} \text{true, if } \exists I \in \mathcal{I}_{p,T}, I' \in \mathcal{I}_{p,T'} : \text{ such that } I \rightarrow I' \\ \text{false, otherwise} \end{cases}$$

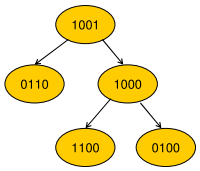
where symbol $\rightarrow$ represents a data dependence. We have an directed edge in PDAG$_p$ from the node that represents $\mathcal{I}_{p,T}$ to the node that represents $\mathcal{I}_{p,T'}$ if and only if $dep(p,T,T')$ holds true.

We now discuss how PDAG$_p$ can be scheduled to reduce energy consumption in disk subsystem. As we discussed earlier, it is important to schedule the iterations in a class one after another. This is not difficult to achieve if we just schedule these iterations such that any two iterations keep their relative orders in the original iteration space traversal (due to our cycle-free assumption). However, as mentioned above, effectiveness of disk power management also depends on the order in which the nodes in PDAG$_p$ are traversed. Specifically, to keep a given disk in the idle state for longer durations of time, we need to select the next node to schedule such that between the two consecutively scheduled nodes, the disks maintain their status as much as possible. Since each node represents a class (a set of iterations) and the tag attached to it indicates us the disks it uses (and the disks that it does not use), one can use this information to select the next node to schedule.

We use a Hamming distance based approach to select the next node to schedule. More specifically, the following observation guides us in selecting a suitable order of scheduling for classes:

> *If $\mathcal{I}_{p,T}$ and $\mathcal{I}_{p,T'}$ are the two nodes (in PDAG$_p$) that are successively visited where T and T' are the respective tags, the variation in disk activation and disk idleness patterns in going from $\mathcal{I}_{p,T}$ to $\mathcal{I}_{p,T'}$ is a function of the Hamming distance between T and T'.*

For instance, in an I/O system with eight disks, if we schedule $\mathcal{I}_{p,01101010}$ and $\mathcal{I}_{p,01100101}$ one after another, the first four disks preserve their states (during this transition), whereas the remaining four disks change their states. Minimizing the Hamming distance between the tags of classes that are visited successively is useful in reducing the disks energy consumption. In other words, for a given disk in the I/O system, in going from one class (node) to another, it is better to keep the states of the disks (active or idle) similar as much as possible. This is because if the first state is 0 and the second is also 0, the disk in question will have a long idle period (which is good from an energy consumption viewpoint); and similarly, if both the states in question are 1, this means that the active



Step	Random	Hamming Based
	$d_0 d_1 d_2 d_3$	$d_0 d_1 d_2 d_3$
1	1001	1001
2	0110	1000
3	1000	1100
4	1100	0100
5	0100	0110

Figure 6: Left: An example PDAG. Right: Two different scheduling.

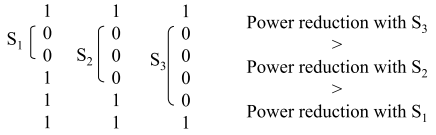


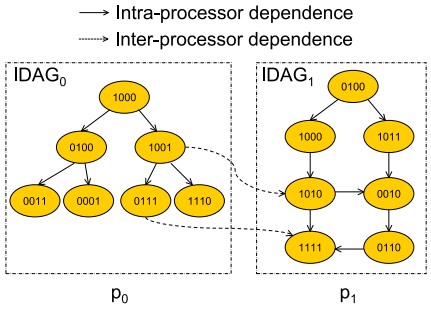
Figure 7: Example optimization schemes (low-power modes).

periods are clustered together; so, we will also have clustered idle periods for the disk (later when we visit the remaining classes). Based on this observation, from the viewpoint of a single processor (p), the problem of reducing disk energy consumption becomes one of scheduling a group of nodes taking accounts of some constraints (inter-class dependences) to minimize (optimize) some objective function (minimizing the Hamming distance between the number of successively visited classes).

To demonstrate how such a scheduling can be beneficial, we consider the PDAG$_p$ shown on the left side of Figure 6. Each node is annotated using its tag (assuming an I/O system with 4 disks). The column titled "Random" on the right side of Figure 6 gives a legal schedule, wherein the next node to be scheduled is selected randomly (by observing the dependences though). Assume that each node takes the same amount of time. Assume further that we have three power optimization schemes that operate as follows (see Figure 7). The first scheme (S_1) is applicable when we have, for a disk, two consecutive "0"s in the schedule (i.e., the same disk is idle in at least two successively-scheduled nodes). The second scheme (S_2) and the third scheme (S_3), on the other hand, are applicable when we have at least three and four consecutive "0"s in the schedule. Based on these power modes, the "Random" scheme can use S_2 for the third disk (d_2) and S_3 for the fourth disk (d_3). In comparison, the last column of the table on the right side of Figure 6 shows the result of our scheduling that minimizes the Hamming distance between the successively-scheduled nodes, as explained above. We see that this schedule is able to use scheme S_1 for disks d_0 and d_1, and scheme S_3 for disks d_2 and d_3, a much better situation compared to the random scheduling case. This small example illustrates how scheduling can impact the opportunities for disk power management.

5.2 Multi-Processor Perspective

An IDAG is constructed from individual PDAGs. One potential problem with the single processor based approach explained above (that operates on individual PDAGs) is that the scheduling is performed for each processor independently. Consequently, while the resulting schedule can appear very good from the perspective of a given processor (as far as reducing disk energy is concerned), when IDAGs are considered together (i.e., the individual schedules are executed in parallel by observing data dependences across processors), they may not perform well. To illustrate this point, let us consider an IDAG for a two processor based system with 4 disks (see Figure 8(a)). As before, each node is annotated using its tag. Let us assume, for simplicity, each node takes the same time (C



(a) An example IDAG constructed from PDAGs of processors p_0 and p_1.

Time	$d_0d_1d_2d_3$		
	p_0	p_1	Usage
1	1000	0100	1100
2	0100	1000	1100
3	1001	1011	1011
4	0011	1010	1011
5	0001	0010	0011
6	0111	0110	0111
7	1110	1111	1111

(b) Scheduling obtained using our algorithm.

Time	$d_0d_1d_2d_3$		
	p_0	p_1	Usage
1	1000	0100	1100
2	1001	1000	1001
3	0100	1010	1110
4	0001	1011	1011
5	0011	0010	0011
6	0111	0110	0111
7	1110	1111	1111

(c) Another legal scheduling.

Figure 8: An example application of our scheduling approach.

cycles) to execute. In Figure 8(c), the columns titled as p_0 and p_1 give the schedules for the two processors (when each schedule is optimized independently as explained in Section 5.1). The last column (marked "Usage"), on the other hand, gives the disk usage when the interleaving effect of these two schedules are taken into account (i.e., each entry in the last column is the result of bit-wise OR of the corresponding entries in the second and third columns). Under the same power-saving schemes assumed above (i.e., S_1, S_2, and S_3 in Figure 7), looking at the "Usage" column, we see that scheme S_1 can be used for disks d_0, d_1, and d_2, and there is no opportunity for applying S_2 or S_3. Figure 8(b) shows the result of our proposed scheduling. This scheduling, whose algorithm will be presented shortly, captures inter-processor effects and results in the disk usage shown in the last column of Figure 8(b). It can be observed that, in this case, we are able to use scheme S_1 for disks d_0, d_2, and d_3 and scheme S_2 for disk d_1.

Our scheduling algorithm for an architecture with P processors and D disks is given in Figure 9. This algorithm takes an IDAG as input, and determines the schedule of nodes for each processor by considering the global (inter-processor) usage of the disks. It uses a D-bit global variable G to represent the current usage of the disks. It schedules a node that is ready to be scheduled for each processor that finishes its current task. At each step, the algorithm first tries to schedule the node whose disk requirement can be satisfied with the current active disks, i.e., we can execute this node without requiring any disks currently in low-power mode. If multiple nodes satisfy this criterion, we select the one that requires the maximum number of disks to make full utilization of the currently active disks. If such a node does not exist, our algorithm schedules the node whose tag is the closest (in terms of Hamming distance) to G, the bit pattern that represents the current disk usage (i.e., the disk usage at that particular point in scheduling). This is to minimize the number of disks whose (active/idle) states need to be changed.

5.3 Node Merging and Node Partitioning

In some cases, an IDAG may contain cycles which prevent a legal traversal (scheduling). We refer to these types of IDAGs as *cyclic IDAGs*. To schedule such graphs, we need to apply some node transformations and eliminate the cycles. An example cyclic IDAG is illustrated on the left side of Figure 10 for an I/O system with 4 disks. Notice that the nodes (classes) with tags 1101, 0001, and 0010 form a cycle, hence the IDAG shown in the figure is not scheduleable. We handle such graphs using two techniques, referred to as *node merging* and *node partitioning* in this paper.[5]

```
|a − b| — Hamming distance between the class {I}_a and {I}_b
last[i] — the last class (node) executed on processor i
schld[i] — set of nodes already scheduled on processor i
can_sch[i] — set of nodes that can be scheduled on processor i
next_time[i] – the time when the current node on processor i is finished
G — global disk usage bit vector
P — number of processors
D — number of disks
| — bit-wise "or"
& — bit-wise "and"
∪ — unordered set union operation
⊕ — ordered set union operation (used for adding a node to set

for i := 0 to P − 1 do {
   next_time[i] = 0;
   last[i] := 0;
}
G := 0;
while(exists unscheduled nodes) do {
   // determine S – the set of processors that are ready
   // to schedule new nodes
   t := ∞;
   for i := 0 to P − 1 do
      if(next_time[i] < t) {
         S := {i};
         t := next_time[i];
      } else if(next_time[i] = t)
         S := S ∪ {i};
   }

   // determine U – the minimum upper boundary of the disk
   // usage after scheduling new nodes
   U := G;
   for each i ∈ S do {
      // X – the set of loop nests that can be scheduled
      // on P_i without turning on new disk
      X := {x|x ∈ can_sch[i] and (d[x]&U) = d[x]};
      if(X = φ)
         X := can_sch[i]; // allowing turning on new disk
      select x ∈ X such that |x − U| is minimized;
      U := U|d[x];
   }

   // schedule nodes on the processors that have
   // finished with the previous nodes
   for each i ∈ S do {
      X := {x|x ∈ can_sch[i] and (d[x]&U) = d[x]};
      select x ∈ X such that |x − U| is minimized;
      // schedule x on P_i;
      schld[i] := schld[i] ⊕ x;
      next_time[i] += t[x];
      last[i] := d[x];
   }

   // determine G – current usage of disks
   G := 0;
   for i := 0 to P do { G := G | last[i]; }

   for i := 0 to P do { update can_sch[i];}
}
```

Figure 9: Proposed scheduling algorithm.

[5] An alternate approach would be constructing the IDAG in a cycle-free manner in the first place. We omit the detailed discussion of this alternative, since the results it generated were very similar to those obtained using node partitioning.

Our first transformation, node merging, combines all the nodes involved in a cycle into a single node. All incident edges on the nodes merged become incident edges on the combined node. The upper right part of Figure 10 illustrates how the nodes that form the cycle in our example can be merged, resulting in a scheduleable (acyclic) IDAG. The important point here is that node merging can be useful even when we do not have any cycles. This is because merging two nodes typically reduces the overhead to be incurred by the generated code and code expansion. One can see this by observing that the number of classes grows exponentially with respect to the number of disks. Therefore, if the underlying disks subsystem has too many disks, we may end up with too many nodes in the IDAG, and for each node we need to generate a different code (as will be explained in the following subsection). In such cases, reducing the number of nodes in the IDAG can be very useful since it reduces the size of the generated code and improves performance. However, a potential drawback of node merging is that the class that represents the combined node accesses in general more disks than the individual classes representing the nodes merged. More specifically, the tag of the combined node is the logical (bit-wise) OR of the tags of the constituent nodes. For example, in Figure 10, the tag of the resulting node is 1111, obtained by bit-wise ORing 1101, 0001, and 0010.

The other technique that can be used for eliminating a cycle in PDAG/IDAG is called node partitioning in this paper. This transformation is in a sense the opposite of node merging, and generates multiple nodes from a single node. To illustrate how it operates, we consider the original cyclic IDAG shown on the left side of Figure 10 again. The lower part of the same figure illustrates the acyclic IDAG obtained by partitioning the node with tag "1101". It is assumed, for illustrative purposes, that after this partitioning, there is a dependence from node "1001" to one of the new nodes, and another dependence from node "0010" to the other new node. Notice that, in the worst case, each of these new nodes inherits the tag of the original node (as in the case of Figure 10). In general, the possibility of node partitioning can be checked as follows. Let us assume $\mathcal{I} = \mathcal{I}_{p,T_1} \cup \mathcal{I}_{p,T_2} \cup ... \cup \mathcal{I}_{p,T_n}$, where $\mathcal{I}_{p,T_1}, \mathcal{I}_{p,T_2}, ..., \mathcal{I}_{p,T_n}$ are the nodes in a cycle. We select a node $\mathcal{I}_{p,T_i}$ and split it into two nodes ($\mathcal{J}_{p,T'}$ and $\mathcal{K}_{p,T''}$) such that all the following constraints are satisfied:

$$\mathcal{J}_{p,T'} \cap \mathcal{K}_{p,T''} = \phi \quad (1)$$

$$\{J \rightarrow K | J \in \mathcal{J}_{p,T'}, K \in \mathcal{K}_{p,T''}\} = \phi \quad (2)$$

$$\{J \rightarrow X | J \in \mathcal{J}_{p,T'}, X \in \mathcal{I} - \mathcal{I}_{p,T_i}\} = \phi \quad (3)$$

$$\{X \rightarrow K | X \in \mathcal{I} - \mathcal{I}_{p,T_i}, K \in \mathcal{K}_{p,T''}\} = \phi \quad (4)$$

Note that, if no node in the cycle can be split with respect to these constraints, we cannot eliminate that cycle by applying node partitioning. In our current implementation, to eliminate a cycle, we first attempt node partitioning. If it does not work, we use node merging.

5.4 Implementation Details

This section gives details of how we generate scheduled code. The main issue here is to generate code for a given class ($\mathcal{I}_{p,T}$). While one can propose an approach employing classical loop transformations such as loop tiling and loop interchange for this purpose, such an approach would not be sufficient, mainly because the iterations that belong to a class may not form a set that can be captured by these (structured) code transformations. Instead, in this study, we use a polyhedral tool called the Omega Library to generate code. The Omega library [22] is a set of routines for manipulating linear constraints over integer variables, Presburger formulas,

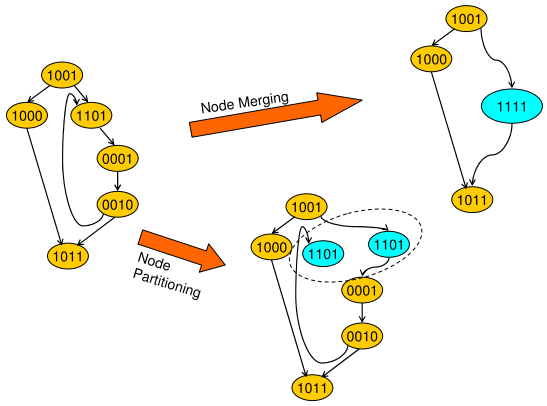


Figure 10: An example that illustrates node merging and node partitioning.

and integer tuple relations and sets. In our context, this library can be used for generating code that enumerates the iterations that belong to a given class. To see this, consider a scenario where a nested loop accesses the arrays stored in an I/O system that consists of two disks (each can hold 45 elements for illustrative purposes). Let us assume that there are two arrays accessed in the nest (U and V), and that the array-to-disk mappings are as follows:

$$d_0 : \quad \{U[i] | 1 \leq i \leq 30\} \cup \{V[i] | 1 \leq i \leq 15\},$$
$$d_1 : \quad \{V[i] | 16 \leq i \leq 30\}.$$

We also assume that the references used in the nest are $U[i]$ and $V[31-i]$, and that the loop iterator (i) takes values between 1 and 29. Using these mappings and references, we can write $\mathcal{I}_{p,10}$, the class that contains iterations which access only d_0, as:

$$\mathcal{I}_{p,10} = \{i | \ (1 \leq i \leq 29) \ \wedge \ (1 \leq i \leq 30) \\ \wedge (1 \leq 31 - i \leq 15) \wedge \neg(16 \leq 31 - i \leq 30)\}.$$

The first constraint in this formulation, $(1 \leq i \leq 29)$, comes from the loop bounds. The second and third constraints ensure that the array elements accessed by U and V fall into the first disk (d_0). Finally, the last constraint guarantees that the elements referenced by V do not reside in the second disk (d_1). By simplifying this set formulation, we obtain:

$$\mathcal{I}_{p,10} = \{i | \ (16 \leq i \leq 29)\}.$$

Then, using the Omega Library's code generator, we can obtain a loop nest that enumerates only these iterations. With a similar analysis, we can also show that:

$$\mathcal{I}_{p,01} = \emptyset \quad \text{and} \quad \mathcal{I}_{p,11} = \{i | \ (1 \leq i \leq 15)\}.$$

6. EXPERIMENTAL SETUP

To generate disk access patterns for our benchmark programs, we designed and implemented a trace generator. This trace generator creates a trace for each processor. The generated trace (which captures parallel disk accesses) is then fed to the simulator. The cycle estimates for the loop nests were obtained from the actual execution of the programs on a SUN Blade1000 machine (UltraSPARC-III architecture operating at 750 MHz with Solaris 2.9) and these estimates were used in all our simulations. In addition to the I/O trace file, the simulator needs the disk layout information for each array, which includes stripe unit size, striping factor (the number

Table 1: Default simulation parameters.

Parameter	Value
Processor Parameters	
Number of Processors	8
Processor Clock Frequency	1.5GHz
Parameters common to TPM and DRPM	
Disk Model	IBM Ultrastar 36Z15
Interface	SCSI
Storage Capacity	18 GB
RPM	15,000
Average seek time	3.4 msec
Average rotation time	2 msec
Internal transfer rate	55 MB/sec
Power (active)	13.5 W
Power (idle)	10.2 W
Power (standby)	2.5 W
Energy (spin down: idle → standby)	13 J
Time (spin down: idle → standby)	1.5 sec
Energy (spin up: standby → active)	135 J
Time (spin up: standby → active)	10.9 sec
Parameters specific to DRPM	
Maximum RPM level	15,000 RPM
Minimum RPM level	3,000 RPM
RPM Step-Size	3,000 RPM
Window Size	250
Striping Information	
Stripe unit (stripe size)	64 KB
Stripe factor (number of disks)	8
Starting iodevice (starting disk)	0

of disks), and starting iodevice (disk). Using disk layout parameters and traces, the simulator determines, for each request, the I/O node(s) that need to be accessed and the duration of access for each I/O node. We assume that each I/O node has one disk and no further striping is applied at the I/O node level, i.e., the data is striped across the I/O nodes only. In our simulator, the striping information is provided from an external file along with other simulation parameters. The default simulation parameters are given in Table 1.

Our disk power simulator, which is similar to DiskSim [5], is driven by externally-provided disk I/O request traces, which are generated, as explained above, by the trace generator. Each I/O request is composed of the following five parameters:

- The id of the processor that issues the request.
- Request arrival time: Time in milliseconds specifying the time at which the disk request arrives.
- Start block number: An integer specifying a logical disk block striped over several I/O nodes.
- Request size: An integer in bytes specifying the size of a request.
- Request type: A character specifying whether the request is a read (R) or a write (W).

Given an I/O trace file, the simulator generates statistical data for performance and energy consumption. Both performance and energy statistics were calculated based on the figures extracted from the data sheet of the IBM Ultrastar 36Z15 [17], and are given in Table 1. The values for power mode transitions are also included in Table 1. Because we are primarily interested in the performance and energy consumption of the disk subsystem, we assume that other performance enhancement techniques like I/O prefetching [4] are not employed. In the rest of the paper, when we say "energy" we mean the energy consumed in the disk subsystem. When we say "execution time/cycles", we mean the time/cycles it takes to complete the application execution. The disk energy consumption includes the energy consumptions in both active and idle periods, taking into account all the states that the disks experience during the entire execution. Also, the performance numbers include all conflicts in accessing the parallel disk system.

Table 2 gives the set of array-based benchmark codes used in this study. These benchmarks were randomly chosen from the Spec2000 floating-point benchmark suite [25]. We made the data manipulated by these benchmarks disk resident. As a result, each array reference causes a disk access unless the data is captured in the buffer cache. Also, to complete our simulations within a reasonable amount of time, we focused only on time-consuming loop nests from these applications. Specifically, from each application, we selected the loop nests whose cumulative I/O times account for at least 90% of the total I/O time of the application using the SUN Analyzer utility. The second column in Table 2 gives the total disk-resident data size manipulated by the selected loop nests, and the third column shows the number of total disk requests made by each application. The last two columns, on the other hand, give the disk energy consumption and execution time, respectively, for each application when *no* disk power management is employed. The energy and performance numbers presented in the rest of this paper are with respect to the values listed in these last two columns of Table 2.

Table 2: Benchmarks and their characteristics.

Name	Data Size(MB)	Number of Disk Reqs	Base Energy(J)	Execution Time(ms)
168.wupwise	176.7	24,718	20835.96	248790.00
171.swim	96.0	3,159	2686.79	32088.98
172.mgrid	384.0	49,152	32759.71	388436.95
173.applu	256.0	32,768	22763.58	270278.68

To compare different approaches to disk power management, we implemented and performed experiments with nine different schemes for each benchmark code in our experimental suite:

- Base: This is the base version that does not employ any power management strategy. *All the reported disk energy and performance numbers are given as normalized values with respect to this version* (see the last two columns of Table 2).
- TPM: This is the traditional disk power management strategy used in studies such as [10] and [11]. In this approach, a disk is spun down after some idleness to save power, and is spun up when a new request arrives. Since the performance cost of spinning up is typically large, TPM can incur significant performance degradations. Also, in order for this scheme to save power, the idleness should be large enough to compensate for the spin-up and spin-down latencies.
- DRPM: This is the dynamic RPM strategy proposed in [14]. Considering the predicted length of the idleness, it sets the rotation speed of the disk to an appropriate level to save power. Therefore, it is effective in saving power even if the idle periods are short. Note that the RPM level used is selected based on the estimated idleness (as in [14]) and we may incur performance penalties, depending on the accuracy of idle time prediction.
- Compiler-directed TPM (C-TPM): This proactive scheme lets the compiler estimate idle periods by analyzing code and considering disk layouts, and then generates TPM power-management calls (`spin_down/up` calls) based on this information.
- Compiler-directed DRPM (C-DRPM): This proactive scheme performs the same estimation of idle periods as in C-TPM, but it generates DRPM power-management calls (`set_rpm` calls). Both C-TPM and C-DRPM are discussed in Section 4.
- Intra-processor TPM (Intra-P-TPM): This corresponds to our code restructuring based approach (from a single-processor perspective) when it is used with C-TPM. The compiler restructures (schedules) code considering disk layout information.
- Intra-processor DRPM (Intra-P-DRPM): This corresponds to our code restructuring based approach (from a single-processor perspective) when it is used with C-DRPM. It uses the same (re-

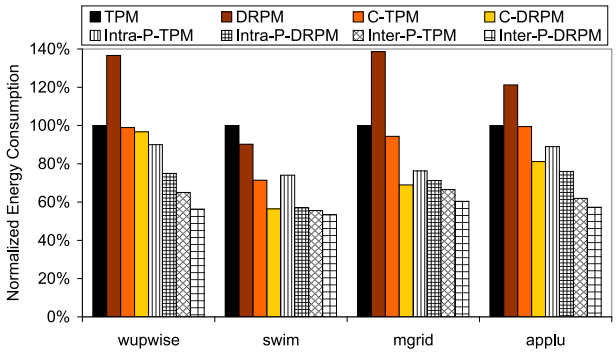


Figure 11: Energy consumptions with different schemes.

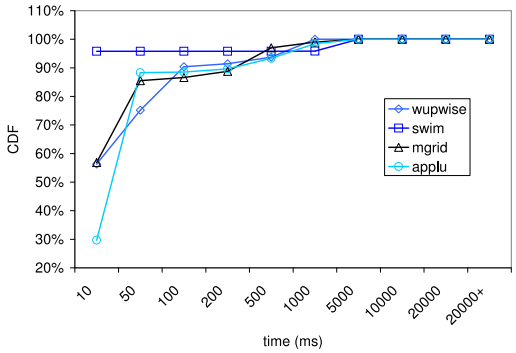


Figure 12: CDF (cumulative distribution function) curves for disk idle times. An (x,y) point on a curve indicates that y% of the idle times has a duration of x (ms) or lower. As mentioned earlier, the minimum amount of idle time required to compensate the cost of spinning down the disk and up (under a TPM-based scheme) is called the threshold. Based on the numbers from IBM Ultrastar 36Z15, the threshold is 15.19 seconds. The results in this graph show that the idle disk times exhibited by these array-based applications are much shorter than the threshold value.

structured) code as in Intra-P-TPM. The scheduling strategy used by Intra-P-TPM and Intra-P-DRPM are explained in Section 5.1.

• Inter-processor TPM (Inter-P-TPM): This corresponds to our code restructuring based approach (from a multi-processor perspective) when it is used with C-TPM. The compiler restructures code considering disk layout information.

• Inter-processor DRPM (Inter-P-DRPM): This corresponds to our code restructuring based approach (from a multi-processor perspective) when it is used with C-DRPM. It uses the same (restructured) code as in Inter-P-TPM. The scheduling strategy used by Intra-P-TPM and Inter-P-DRPM are explained in Section 5.2.

Note that the only modification to the input code made by C-TPM and C-DRPM are the insertions of explicit power management calls (which are then simulated by the disk simulator). In comparison, Intra-P-TPM, Intra-P-DRPM, Inter-P-TPM and Inter-P-DRPM restructure the application code using scheduling. The necessary code modifications for these schemes are automated using the SUIF infrastructure [15], with the help of Omega Library [22] as has been discussed earlier. As a result of these compiler transformations, we observed that the original compilation times were almost doubled. We believe that, considering the large benefits of the approach, this increase in compilation times is tolerable.

7. EXPERIMENTAL RESULTS

The graph in Figure 11 gives the energy consumptions of our benchmarks under the different schemes explained above. One can make several observations from these results. First, the TPM scheme does not achieve any disk energy savings since most of the disk idle times in these applications are not very large as shown in Figure 12. And, for very few relatively long idle periods, the TPM scheme fails to exploit them as well, mainly because it waits for some time (at the beginning of each idle period) before spinning down the disk (see Figure 3). In comparison, C-TPM brings about 9% energy savings by taking advantage of these few relatively long idle periods, which demonstrates the benefits brought by proactive disk power management. The second observation is that the DRPM scheme consumes more energy than the original version (Base), due to poor estimation of idle periods. However, its proactive version (C-DRPM) achieves nearly 24% disk energy savings on average. Our third observation is that the best results for all applications are obtained with the Inter-P-TPM and Inter-P-DRPM versions. Specifically, they achieve, respectively, about 38% and 43% savings in disk energy. In comparison, Intra-P-TPM and Intra-P-DRPM save around 18% and 30% disk energy, respectively. In other words, capturing and exploiting interprocessor disk access pattern is critical in maximizing savings. In fact, Inter-P-TPM generates better energy savings on average than Intra-P-DRPM, meaning that using a less powerful architectural mechanism with more sophisticated code restructuring generates better results than employing more powerful architectural mechanism with less sophisticated code restructuring for this set of applications.

It is to be noted however that the energy consumption is just one part of the big picture. To have a fair comparison between the different schemes tested, one needs to consider their performances (i.e., execution times/cycles) as well. The bar-chart in Figure 13 gives the normalized execution times (with respect to the base version) for the different schemes evaluated. One can observe that only the DRPM version incurs some performance penalty, 70% on average across our four benchmarks. The reason why TPM does not incur any performance penalty is that it is not generally applicable, given the short disk idle times as discussed earlier. We also see that all the compiler-directed schemes, C-DRPM, C-TPM, Intra-P-TPM, Intra-P-DRPM, Inter-P-TPM, and Inter-P-DRPM, incur almost no performance penalty. The main reason for this is that these schemes start to bring the disk to the desired RPM level before it is actually needed (using preactivation), and the disk becomes ready when the access takes place. This is achieved by accurate prediction of disk idle periods for the application domain we target. These results, along with those presented in Figure 11, indicate that the compiler-guided proactive disk power management and code restructuring can be very useful in practice, in terms of both disk energy consumption and execution time penalty, and the best savings are achieved by our code restructuring approach. Note that, since our compiler approach does not increase execution times, it does not cause much extra power consumption on other system components. The only additional energy overhead is due to execution of the inserted power management calls (instructions); but, we found this cost to be negligible.

In the rest of our experimental analysis, we vary the values of some of the simulation parameters, and study their impacts on energy consumption. We do not present any further performance data, mainly because, except for the DRPM scheme, none of the schemes evaluated causes any substantial increase in original execution cycles. More specifically, except for DRPM, the average execution

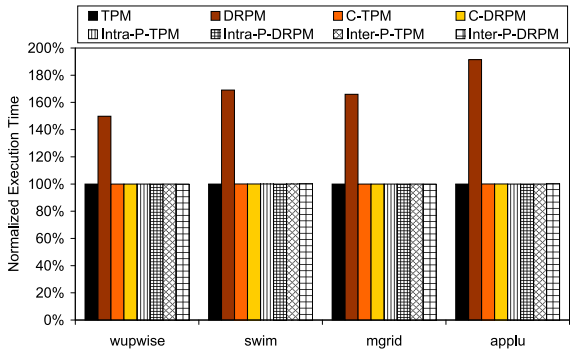


Figure 13: Execution cycles with different schemes.

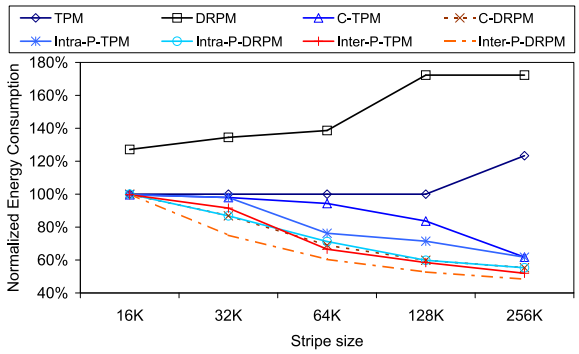


Figure 14: Impact of stripe size on energy consumption.

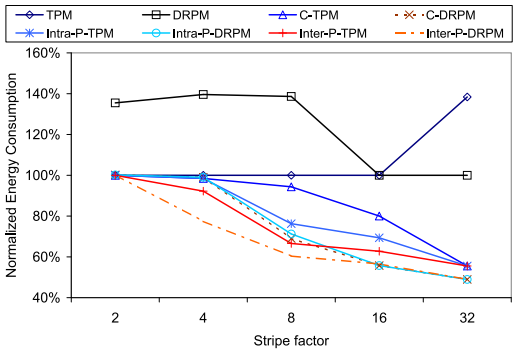


Figure 15: Impact of stripe factor on energy consumption.

time increase was always less than 1%. We focus on two important parameters in our sensitivity analysis: disk layout and number of processors. Since disk layout has three components as explained in Section 3, we study each of them separately. In each experiment, we change the value of only one parameter; the rest of the simulation parameters use their default values listed in Table 1. Also, since our results with different benchmarks resulted in similar trends and observations, we present the result for the mgrid benchmark only.

Figure 14 gives the normalized energy consumptions with the different stripe sizes. We see from these results that the energy savings brought by the compiler-based schemes increase as the stripe size increases. This can be explained as follows. When the stripe size is very small (16K), disks do not experience much idleness. In fact, the disk idleness in this case becomes so small that even code restructuring cannot take advantage of it. When the stripe size is increased, more disk requests can be serviced by a single stripe, i.e., the stripe-level data reuse improves. As a result, the compiler-based approaches have more opportunities for optimization, which in turn helps reduce disk energy consumption. We see from Figure 14 that Inter-P-DRPM generates the best savings with 256K stripe size, and the difference between it and the DRPM scheme reaches its peak at this value. The next parameter whose variation we study is the stripe factor (the number of disks). Recall that the default number of disks used so far in our experiments was 8. Figure 15 gives the normalized energy values under different stripe factors. An observation we can make from these curves is that, when we have only two disks, there is not much opportunity for power saving (due to lack of disk idleness), and (except for DRPM) all the schemes behave similarly. As the number of disks is increased, disk idleness increases, and consequently, the compiler schemes exhibit a better behavior. When the number of disks is very high (32), the disk idleness reaches a very high level, and one may not need sophisticated code restructuring in this case (for our particular data set sizes). In fact, at this point, all the TPM-based compiler schemes behave similarly, and all the DRPM-based compiler schemes behave similarly.

We next study how the starting disk used for striping could affect the results. To perform this set of experiments, we generated a random integer number (for each array in the mgrid benchmark) between 1 and 8 to select the disk from which the array is striped. The results for five such experiments are presented in Figure 16. We see from these results that the general trends (and our savings) are very similar across these different layouts. This indicates that the starting disk (for striping) may not be a very important factor as far as our compiler-based schemes are concerned.

The last parameter whose variation we study is the number of processors. Figures 17 gives the normalized energy results with different processor counts. As before, all other parameters are set to their default values given in Table 1. One can see from these results that the effectiveness of the compiler-directed code restructuring is consistent across the different processor counts. The reason that the C-TPM and C-DRPM schemes do not behave very well with the large number of processors is the difficulty in inserting explicit power management calls, due to small iteration counts with large number of processors. This problem does not usually exist in the code restructuring based schemes since they cluster idle and active periods.

8. CONCLUDING REMARKS

Excessive power consumption is becoming a major barrier to extracting the maximum performance from high-end parallel systems. Therefore, techniques oriented towards reducing power consumption of such systems are expected to become increasingly important in the future. Since disk subsystems of parallel architectures are known to consume a large fraction of the overall power budget, they are an important optimization target. Unfortunately, most of the prior work on disk power management focused exclusively on hardware-based approaches that operate with past history information collected during execution. In contrast, this paper proposes a compiler-driven approach to disk power management for data-intensive scientific applications. The compiler in our approach derives data access pattern and, by combining this information with disk layout of array data, it obtains the disk access pattern. This paper demonstrates two ways of utilizing disk access patterns: proactive disk power management and code restructuring for reducing disk power consumption. Our experimental analysis with several applications that operate with disk-resident data sets are very promising and show that the proposed compiler-driven approach performs much better than existing hardware-based techniques.

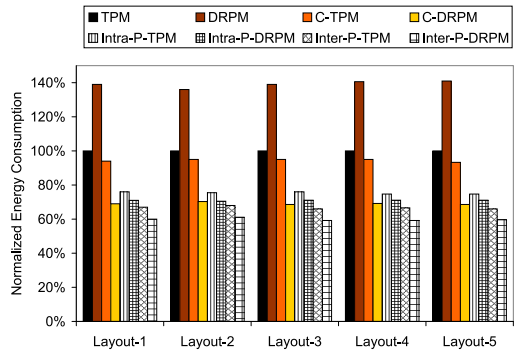


Figure 16: Impact of starting disk on energy consumption.

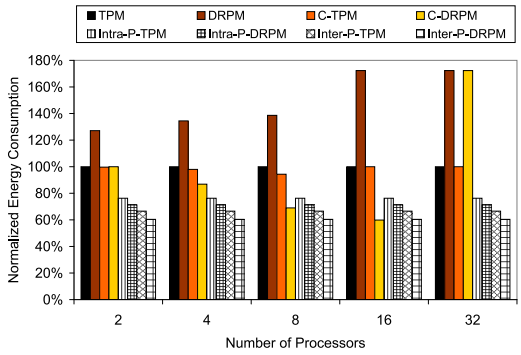


Figure 17: Impact of processor count on energy consumption.

9. REFERENCES

[1] Where are all the green computers? http://environment.about.com/od/greenschoolsupplies/a/computers.htm, 2004.

[2] J. M. Anderson. *Automatic Computation and Data Decomposition for Multiprocessors*. Ph.d thesis, Stanford University, 1997. CSL-TR-97-719.

[3] L. Benini, G. D. Micheli, E. Macii, M. Poncino, and R. Scarsi. Symbolic Synthesis of Clock-Gating Logic for Power Optimization of Synchronous Controllers. *ACM Trans. Des. Autom. Electron. Syst.*, 4(4):351–375, 1999.

[4] A. D. Brown, T. C. Mowry, and O. Krieger. Compiler-Based I/O Prefetching for Out-Of-Core Applications. *ACM Transactions on Computer Systems*, 19(2):111–170, May 2001.

[5] J. S. Bucy, G. R. Ganger, and Contributors. The Disksim Simulation Environment Version 3.0 Reference Manual. Technical Report CMU-CS-03-102, CMU, January 2003.

[6] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving Disk Energy in Network Servers. In *Proceedings of the 17th International Conference on Supercomputing*, pages 86–97. ACM, June 2003.

[7] J. Chase, D. Anderson, P. Thackar, A. Vahdat, and R. Boyle. Managing Energy and Server Resources in Hosting Centers. In *Proc. of the 18th Symposium on Operating Systems Principles*, pages 103–116, October 2001.

[8] J. Chase and R. Doyle. Balance of Power: Energy Management for Server Clusters. In *Proc. of the 8th Workshop on Hot Topics in Operating Systems*, page 165, May 2001.

[9] X. Chen and L. Peh. Leakage Power Modeling and Optimization in Interconnection Networks. In *Proc. of the International Symposium on Low Power and Electronics Design*, pages 90–95, August 2003.

[10] F. Douglis, P. Krishnan, and B. Bershad. Adaptive Disk Spin-Down Policies for Mobile Computers. In *Proc. of the 2nd Symposium on Mobile and Location-Independent Computing*, pages 121–137, 1995.

[11] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the Power-Hungry Disk. In *Proc. of the USENIX Winter Conference*, pages 292–306, 1994.

[12] M. Elnozahy, M. Kistler, and R. Rajamony. Energy-Efficient Server Clusters. In *Proc. of the Second Workshop on Power Aware Computing Systems*, February 2002.

[13] M. Elnozahy, M. Kistler, and R. Rajamony. Energy Conservation Policies for Web Servers. In *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

[14] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: Dynamic Speed Control for Power Management in Server Class Disks. In *Proc. of the International Symposium on Computer Architecture*, pages 169–179, June 2003.

[15] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *Computer*, 29(12):84–89, December 1996.

[16] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini. Application Transformations for Energy and Performance-Aware Devicement Management. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 121–130. IEEE, Sep 2002.

[17] IBM. *Ultrastar 36ZX & 18LZX*, 1999.

[18] R. K. K. Li, P. Horton, and T. Anderson. A Quantitative Analysis of Disk Drive Power Management in Portable Computers. In *Proc. of the USENIX Winter Conference*, pages 279–292, 1994.

[19] E. J. Kim, K. H. Yum, G. Link, C. R. Das, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Energy Optimization Techniques in Cluster Interconnects. In *Proc. of International Symposium on Low Power Electronics and Design*, pages 459–464. ACM, August 2003.

[20] C. H. Koelbel, D. B. Loveman, and R. S. Schreiber. *The High Performance Fortran Handbook*. The MIT Press, 1993.

[21] M. Pedram. Power Optimization and Management in Embedded Systems. In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 239–244. ACM Press, 2001.

[22] W. Pugh. A Practical Algorithm for Exact Array Eependency Analysis. *Comm. of the ACM*, 35(8):102–114, August 1992.

[23] R. B. Ross, P. H. Carns, W. B. L. III, and R. Latham. Using the Parallel Virtual File System, July 2002.

[24] T. Simunic, L. Benini, and G. D. Micheli. Energy-Efficient Design of Battery-powered Embedded Systems. In *ISLPED '99: Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, pages 212–217. ACM Press, 1999.

[25] SPEC. Specfp 2000, 2000.

[26] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.