Compiler-Directed Energy Optimization for Parallel-Disk-Based Systems

Seung Woo Son, *Student Member*, *IEEE*, Guangyu Chen, *Student Member*, *IEEE*, Ozcan Ozturk, *Student Member*, *IEEE*, Mahmut Kandemir, *Member*, *IEEE*, and Alok Choudhary, *Fellow*, *IEEE*

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. Although 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 (for example, 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 the disk access pattern, i.e., the order in which parallel disks are accessed. This paper demonstrates two uses of this information. First, we can implement 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 the length of idle disk 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 Spec 2000 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, although proactive disk power management is very effective, code restructuring for disk power achieves additional energy savings across all the benchmarks tested, and these savings are very close to optimal savings that can be obtained through an Integer Linear Programming (ILP)-based scheme.

Index Terms—Disk subsystem, I/O traces, optimizing compilers, power-aware computing, parallel I/O.

1 Introduction and Motivation

Power consumption is becoming a growing 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 [11], [12]. Second, computing servers typically contribute a large fraction of the overall power budgets of institutions and even cities [7], [5], [6]. 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 [13], [14], [5], [7], [11] 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 [9], [10], [22] or rotating disks with reduced speed [13], [5] when some amount of latency can be tolerated. A review of the prior work on disk power management is given in Section 2. Although such techniques have been shown to be effective in certain cases, they have a common drawback: They operate in a *reactive* manner, i.e., they control disk behavior based on observed disk activity (for example, idle and active periods). In practice, this can cause two problems. First, they may fail to select the most appropriate disk power management scheme since their disk idleness estimations 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 environments with parallel disks. An optimizing compiler is in a very good position for the application domain and execution platform stated above. This is because the compiler can analyze the data access pattern of a scientific application based on a high-level representation of the program [37], [23], which enables us to capture how the disk-resident data are accessed and shared by parallel processors. As for determining disk idle and active periods, extracting the 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

- *S.W. Son, O. Ozturk, and M. Kandemir are with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802. E-mail: {sson, ozturk, kandemir}@cse.psu.edu.*
- *G. Chen is with Microsoft, One Microsoft Way, Redmond, WA 98052. E-mail: guchen@microsoft.com.*
- *A. Choudhary is with the Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208. E-mail: choudhar@ece.northwestern.edu.*

Manuscript received 22 Nov. 2005; revised 19 Oct. 2006; accepted 1 Nov. 2006; published online 9 Jan. 2007.
Recommended for acceptance by X. Zhang.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0485-1105.
Digital Object Identifier no. 10.1109/TPDS.2007.1056.

information of disk-resident data to the compiler. In other words, the proposed compiler support obtains the disk access pattern by using the data access pattern and disk layout information for array data. Section 3 explains the proposed disk access pattern extraction process 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 (for example, 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 power management strategies. Second, the compiler can restructure the given application code to increase idle periods of disks, thereby allowing a more effective disk power management. We demonstrate that this code restructuring can be expressed as a *scheduling problem*, which, in turn, can be handled by any known heuristic or exact scheduling algorithm. 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 interprocessor 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 [34] with disk-resident data sets show that, although 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 increase with the 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. In addition, we compare our approach to an optimal scheme implemented using Integer Linear Programming (ILP) and show that our savings come very close to optimal savings.

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 (nontraditional) 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 have been a significant number of past work on power management of high-end computing systems [6], [20], [8] and low-end embedded devices [26], [31], [3]. Due to space concerns, we limit ourselves, in this section, to disk energy optimization-related studies.

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 traditional power management (TPM) [9], [22], [10] 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 an 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. In addition to spinning down the spindle of the disk, disk drive manufacturers employ several other techniques that incur less spin-up penalty while reducing power. These techniques include slowing or stopping the embedded processor in the disk controller, turning off the servo system that controls the disk arm, and placing the head onto a landing position [18], [17]. Although TPM is a good mechanism for conserving disk power in laptop systems and embedded environments, recent studies (for example, [13] and [14]) show that it is not a preferable option in the server or cluster domains for two reasons. First, the access patterns in server workloads are mainly small and noncontiguous and, consequently, disk idle times are not long enough to accommodate TPM. Second, for performance reasons, server class disks are operated at very high revolutions per minute (RPM), typically above 10,000 RPM, and the disk spin-up/down times are very long, which, in turn, makes the threshold value very large.

Since exploiting idle time is hardly a viable option for the server-class disks, Gurumurthi et al. [13] 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 more power it consumes. The disk that employs the DRPM technique is already commercially available [17], though it is not a full-blown one. 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 the literature [36], [28], [27] 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 a 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 in the prior research [13] that DRPM can save a significant amount of disk power by exploiting even small idle times, and it incurs relatively small performance penalty compared to TPM. A similar technique based on modulating between only two disk speeds has been proposed and evaluated in [5]. In the rest of this paper, the term "low-power mode" (or "low-power state") refers to either a disk that 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). Although several

other mechanisms such as turning off the disk controller or unloading the head from the media can be employed as an alternative power management mechanism, these options are not our focus because our goal is not to propose a new hardware disk power management technique. The focus of our approach instead 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-based and DRPM-based I/O systems.

Several studies have focused on disk power management at the operating system (OS) layer and application/compiler layer. The efforts at the OS layer include energy-efficient caching and prefetching [25], power-aware least recently used (PA-LRU) algorithm [40], and partition-based least recently used (PB-LRU) algorithm [41]. The motivation of these works stems from the fact that the idle periods generated by conventional caching and prefetching techniques are not long enough to place idle disks into low-power modes. Instead of spreading disk accesses across the entire execution period, energy-efficient prefetching generates burst disk access patterns, which are more desirable from the energy-saving angle. On the other hand, the PA-LRU and PB-LRU techniques increase disk idle periods by selectively maintaining cache blocks from certain disks, thereby increasing chances for the idle disks to remain in low-power modes for a longer period of time. More recently, Zhu et al. [39] proposed a holistic disk power management technique, called Hibernator, that combines three different techniques: dynamic disk speed setting, multitier data layout, and data reorganization. Rather than modulating disk speed at a fine granularity, their idea is to adjust disk speed at coarse granularity, which is preferable from the disk reliability perspective. In an effort to save disk energy consumption at the compiler layer, Heath et al. [16] propose an application code transformation technique for energy/performance-aware device management on laptop disks by utilizing available buffer space. Our approach is different from their work in that we focus on power management on parallel disk systems, which exhibit entirely different idle period patterns. In addition, since they target laptop-based environments, they use a 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 envision it coexisting 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 important characteristic 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. Before describing our disk access pattern mechanism, we would like to mention briefly that, even for real applications that show mixed access patterns, our scheme can still be useful for analyzing and predicting the behavior of the code portion where the access patterns are regular. However, the energy savings we would achieve can be reduced, depending on the amount of the code portion whose access patterns are irregular.

One requirement for being able to use a compiler in reducing disk power consumption is to capture how parallel disks are accessed at a high level (that is, source code level). We use the term *disk access pattern* in this paper to refer to 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 granularity. To obtain this information, the compiler needs the data access pattern of the application code being optimized and disk layout information for the array data (see Fig. 1a). 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 (for example, 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 are striped across the parallel disks and can optimize the source code accordingly.

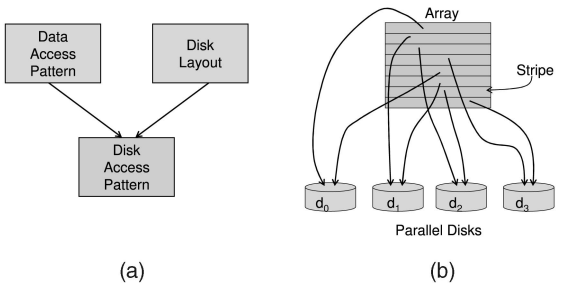


Fig. 1. (a) Determining disk access pattern. (b) Striping an array over four disks.

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 Fig. 1b). This permits multiple processes to access different portions of the data concurrently without much disk contention. Although striping can be performed manually, many file systems today provide automatic support for it, as will be explained below. In this work, we represent a disk layout of an array using a triplet of the form

$$(\text{starting\_disk}, \text{stripe\_factor}, \text{stripe\_size}).$$

The first component in this triplet indicates the disk from which the array has 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 the disk layout information when the file is created. For example, in the Parallel Virtual File System (PVFS) [30], 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 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

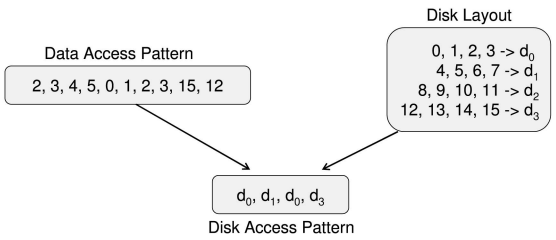


Fig. 2. A data access pattern and the corresponding disk access pattern. $< d_i, t_j >$ means that disk d_i is (estimated to be) used for t_j cycles.

access pattern it extracts to determine the disk access pattern. On the other hand, if the file is already created on the disk system and we want to obtain its layout, we can use `pvfs_ioctl()` call along with the `pvfs_filestat` structure, which fills in a data structure with the corresponding disk layout. The obtained disk layout of the file in question can then be passed to the compiler as a command line parameter.

An important point to note is that we assume each data array manipulated by the application is stored in a separate file in the I/O system. Since each file can have a different triplet of the kind shown above, each array can have a different disk layout than the others. Clearly, one can also optimize disk layout itself in an energy-efficient way without incurring any degradation in disk performance, that is, one can try to come up with a scheme that determines the optimal number of disks to satisfy both energy and performance constraints. Although combining our code-restructuring algorithm with schemes that determine energy-efficient disk layouts (such as that in [32]) 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 (as explained above), which subsequently uses it for determining the disk access patterns.

Fig. 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 12 elements of an array are distributed (striped) across four disks (d_0 through d_3). In the disk access pattern, a $\langle d_i, t_j \rangle$ 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).

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 (for example, TPM versus DRPM). For TPM disks, we use `spin_up()` and `spin_down()` calls. The format of the `spin_down()` call is given as follows:

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

where d_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 (that is, 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

$$Q_{su} = \left\lceil \frac{T_{su}}{s + T_m} \right\rceil, \quad (1)$$

where Q_{su} is the preactivation distance (in terms of loop iterations), T_{su} is the expected spin-up time (in cycles), 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 given 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 disk preactivation is to eliminate this performance penalty. Although our discussion so far has focused on 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 collected disk access pattern 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 jth RPM level (that is, 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 levels involved [13], 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

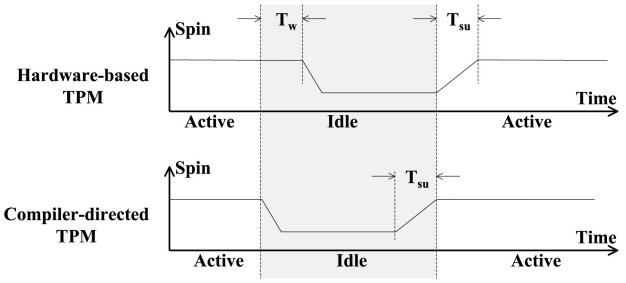


Fig. 3. Comparison of the hardware-based TPM and the proposed compiler-directed TPM. In the hardware-based scheme, period T_w is for detecting idleness, and T_{su} is the spin-up latency. The compiler-directed scheme can eliminate the impact of both these latencies.

probability), the information it extracts (regarding disk idle/active times) may not be 100 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, although 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 Section 5 in detail.

Fig. 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. Second, the compiler-directed TPM can avoid the performance overhead, using preactivation, due to the spin-up latency when an idle disk is accessed. Fig. 4 presents our compiler algorithms for disk energy optimization for the TPM case. Our algorithm works in two steps. In the first step, we build a *Loop Transition Graph* (LTG) for a given procedure.[1] 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 the 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. Fig. 5b shows an LTG for the code fragment in Fig. 5a. In the second step, our algorithm inserts a code to the program to spin the disks up/down. Specifically, for each node L_i in the 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 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 performing this transformation, we hide the performance overhead due to disk spin up. That is, as explained earlier, this preactivation eliminates potential performance penalty. Fig. 5c is the transformed code fragment after applying our algorithm.

1. Our current implementation is applied to each procedure separately; that is, we do not perform any interprocedural 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)";
    }
  }
}
```

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

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 disk power management scheme discussed above. This code restructuring approach operates on a graph representation called the *Interprocessor 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.

We assume that the set of loop iterations that will be executed by each processor has already been determined prior to the approach. For this purpose, either user-assisted (for example, [21]) or compiler-directed (for example, [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 loop 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 as 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 dth position of T $(0 \leq d \leq D-1)$ is 1 if and only if loop iteration I

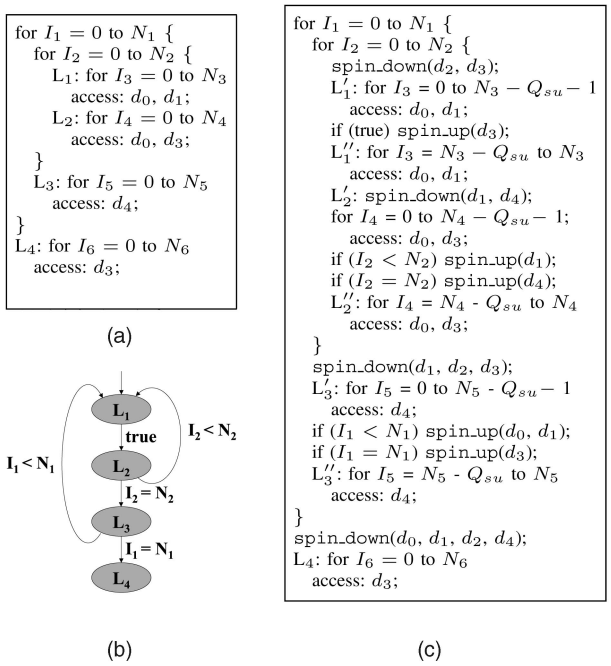


Fig. 5. An example that illustrates proactive disk power management. (a) Original code fragment. (b) LTG for the code fragment in (a). Nodes L_1, L_2, L_3, and L_4 correspond to the loop nests with labels L_1, L_2, L_3, and L_4, respectively. (c) Transformed code fragment. The loops L_1, L_2, and L_3 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}.

accesses disk d.[2] 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 an input. Now, we can classify the loop 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

$$\mathcal{I}_{p,T} = \{I \quad | \quad I \in \mathcal{I}_p \quad \wedge \quad tag(I) = T\}, \tag{2}$$

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

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 a single processor's perspective and multiprocessors' perspective, respectively. What we mean by "scheduling" in this context is determining 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 the 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.

2. 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.

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:

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

where symbol $\rightarrow$ represents a data dependence. We have a 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.

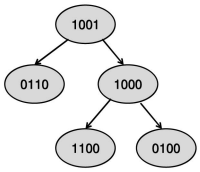
We now discuss how PDAG_p can be scheduled to reduce energy consumption in a disk subsystem. As we discussed earlier, it is important to schedule the loop iterations in a class one after another. This is not difficult to achieve if we just schedule these iterations such that any two iterations keep their relative orders in the original iteration space traversal (due to our cycle-free assumption). However, as mentioned above, the 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 successively 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 gives 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 their 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 would preserve their states (during this transition), whereas the remaining four disks would 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 set of disks 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 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 into account of certain constraints (interclass dependences) to minimize (optimize) a given 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 example PDAG_p shown in Fig. 6a. Each node



(a)

Step	Random	Hamming Based
	$d_0d_1d_2d_3$	$d_0d_1d_2d_3$
1	1001	1001
2	0110	1000
3	1000	1100
4	1100	0100
5	0100	0110

(b)

Fig. 6. (a) An example PDAG. (b) Two different scheduling approaches.

is annotated using its tag (assuming an I/O system with four disks). The column titled "Random" in Fig. 6b 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 Fig. 7). The first scheme (S_1) is applicable when we have, for a disk, two consecutive 0s in the schedule (that is, 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 0s, respectively, 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 in Fig. 6b 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, resulting in a much better behavior compared to the random scheduling case. This small example illustrates how scheduling can impact the opportunities for disk power management.

5.2 Multiprocessor 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, although 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 four disks (see Fig. 8a). As before, each node is annotated using its tag. Let us assume, for simplicity, that each node takes the same time (C cycles) to execute. In Fig. 8c, the columns titled 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

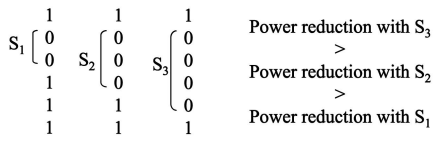
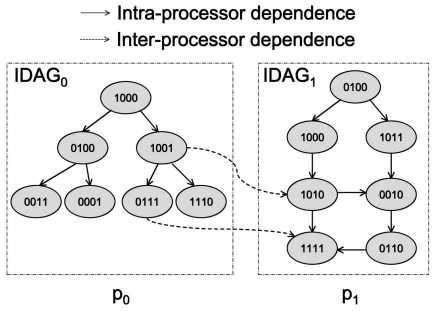


Fig. 7. Example optimization schemes (low-power modes).



(a)

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)

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)

Fig. 8. An example application of our scheduling approach. (a) An example IDAG constructed from PDAGs of processors p_0 and p_1. (b) Scheduling obtained using our algorithm. (c) Another legal scheduling.

account (i.e., each entry in the last column is the result of the bitwise OR of the corresponding entries in the second and third columns). Under the same power-saving schemes assumed above (that is, S_1, S_2, and S_3 in Fig. 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. Fig. 8b shows the result of our proposed scheduling. This scheduling, whose algorithm will be presented shortly, captures interprocessor effects and results in the disk usage shown in the last column of Fig. 8b. 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 Fig. 9. This algorithm takes an IDAG as input and determines the schedule of nodes for each processor by considering the global (interprocessor) 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 set of active disks, i.e., we can execute this node without requiring any disks that are 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 (that is, the disk usage at that particular point during 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 that prevent a legal traversal (scheduling). We refer to these types of IDAGs as *cyclic IDAGs*. To schedule such graphs, we may need to apply node transformations and eliminate the cycles. An example cyclic IDAG is illustrated on the left side

```
|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];}
}
```

Fig. 9. Proposed scheduling algorithm.

of Fig. 10 for an I/O system with four disks. Notice that the nodes (classes) with tags 1101, 0001, and 0010 form a cycle, and thus, the IDAG shown in the figure is not schedulable. In our framework, we handle such graphs using two techniques, referred to as *node merging* and *node partitioning*.[3]

Our first transformation, node merging, combines all the nodes involved in a cycle into a single node. All of the incident edges on the nodes that merged become incident edges on the combined node. The upper right part of Fig. 10 illustrates how the nodes that form the cycle in our example can be merged, resulting in a schedulable (acyclic) IDAG. The important point to note 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

3. 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.

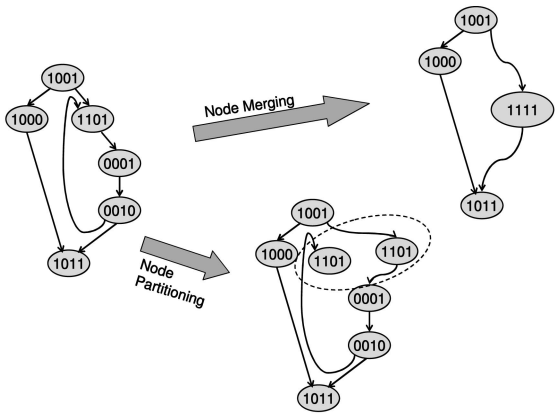


Fig. 10. An example that illustrates node merging and node partitioning.

with respect to the number of disks. Therefore, if the underlying disk 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 Section 5.4). In such cases, reducing the number of nodes in the IDAG can be very useful since it helps reduce 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 merged nodes. More specifically, the tag of the combined node is the logical (bitwise) OR of the tags of the constituent nodes. For example, in Fig. 10, the tag of the resulting node is 1111, obtained by bitwise 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 Fig. 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 in Fig. 10). In general, the possibility of node partitioning can be checked as follows: Let us assume $\mathcal{I} = \mathcal{I}_{p,T_1} \cup \mathcal{I}_{p,T_2} \cup \ldots \cup \mathcal{I}_{p,T_n}$, where $\mathcal{I}_{p,T_1}, \mathcal{I}_{p,T_2}, \ldots, \mathcal{I}_{p,T_n}$ are the nodes involved 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:

$$\mathcal{J}_{p,T'} \cap \mathcal{K}_{p,T''} = \phi,$$
$$\{J \rightarrow K | J \in \mathcal{J}_{p,T'}, K \in \mathcal{K}_{p,T''}\} = \phi,$$
$$\{J \rightarrow X | J \in \mathcal{J}_{p,T'}, X \in \mathcal{I} - \mathcal{I}_{p,T_i}\} = \phi,$$
$$\{X \rightarrow K | X \in \mathcal{I} - \mathcal{I}_{p,T_i}, K \in \mathcal{K}_{p,T''}\} = \phi.$$

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 the scheduled code. The main issue here is to generate a code for a given class ($\mathcal{I}_{p,T}$). Although 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 regular set that can easily 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 [29] provides a set of routines for manipulating linear constraints over integer variables, Presburger formulas, and integer tuple relations and sets. In our context, this library can be used for generating a code that enumerates the loop 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:

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

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 that access only d_0, as

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

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

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

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

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

6 Experimental Setup

6.1 Setup and Benchmarks

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 of disks), and starting disk. Using the 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, that is, 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.

TABLE 1
Default Simulation Parameters

Parameter	Value
Processor & Disk Parameter	
Number of Processors	8
Processor Clock Frequency	1.5GHz
Disk Model	IBM Ultrastar 36Z15
Interface	SCSI
Capacity	18.4 GB
RPM	15,000
Average seek time	3.4 ms
Average rotational latency	2.0 ms
Internal transfer rate	55 MB/s
Power Model	
Power (active)	13.5 W
Power (idle)	10.2 W
Power (standby)	2.5 W
Energy (spin down: idle $\rightarrow$ standby)	13 J
Time (spin down: idle $\rightarrow$ standby)	1.5 sec
Energy (spin up: standby $\rightarrow$ active)	135 J
Time (spin up: standby $\rightarrow$ active)	10.9 sec
DRPM & Striping Parameter	
Maximum RPM level	15,000 RPM
Minimum RPM level	3,000 RPM
RPM Step-Size	3,000 RPM
Window Size	250
Stripe size	64 KB
Stripe factor	8
Starting disk	0

Our disk power simulator, which is similar to DiskSim [4], 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:

- *ID*. The ID of the processor that issues the request.
- *Request arrival time.* Time (in milliseconds) specifying the time at which the disk request arrives.
- *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) type.

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 [19] and are given in Table 1. The values for power mode transitions are also included in Table 1. 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 all 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
Benchmarks and Their Characteristics

Name	Data Size(GB)	Number of Disk Reqs	Base Energy(J)	Execution Time(sec)
168.wupwise	88.6	1,384,208	145851.72	1741.53
171.swim	64.7	1,010,880	107440	1283.56
172.mgrid	75.5	1,179,648	98279.13	1165.31
173.applu	100.6	1,572,864	136581.48	1621.67

Table 2 gives the set of array-based benchmark codes used in this study. These benchmarks were randomly chosen from the Spec 2000 floating-point benchmark suite [34]. As none of the original Spec 2000 requires a memory footprint larger than 200 Mbyte, we increased the data set size by manipulating the dimension sizes of the arrays and the corresponding loop bounds accordingly. We also made the data manipulated by these benchmarks disk resident by mapping each array data to the corresponding file stored in the disk subsystem. As a result, each array reference causes a disk access unless the data is captured in the buffer cache. However, to be fair in our evaluation, we hand optimized the I/O behavior of these applications as much as we could. In other words, even the original versions of these applications do *not* perform any unnecessary disk I/O. 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 percent of the total I/O time of the application using the SUN Analyzer utility [35]. 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.

6.2 Versions

To compare the different approaches to disk power management, we implemented and performed experiments with nine 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 [9] and [10]. 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 DRPM strategy proposed in [13]. 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 [13]), 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 the necessary 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 explicit DRPM power-management calls (`set_rpm` calls). Both C-TPM and C-DRPM are discussed in Section 4.
- Intraprocessor 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) the code considering the disk layout information.
- Intraprocessor 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 (restructured) code as in Intra-P-TPM. The scheduling strategy used by Intra-P-TPM and Intra-P-DRPM are explained in Section 5.1.
- Interprocessor TPM (Inter-P-TPM). This corresponds to our code-restructuring-based approach (from a multiprocessor perspective) when it is used with C-TPM. The compiler restructures code considering disk layout information.
- Interprocessor DRPM (Inter-P-DRPM). This corresponds to our code-restructuring-based approach (from a multiprocessor 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 insertions of the explicit disk power management calls, which are then simulated by the disk simulator. In comparison, the Intra-P-TPM, Intra-P-DRPM, Inter-P-TPM, and Inter-P-DRPM schemes restructure the application code using scheduling. The necessary code modifications for these schemes are automated using the Stanford University Intermediate Compiler (SUIF) infrastructure [15], with the help of Omega Library [29], 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.

- ILP-based Scheme. In addition to the schemes explained so far, we also implemented the optimal scheme in order to demonstrate how close our approach comes to the optimal solution. Specifically, we implemented an ILP-based approach to determine the maximum possible energy savings for a given disk trace. Although other methods such as a direct calculation of optimal disk power management scheme from the given disk traces or those based on genetic algorithms can also be used, we found ILP

TABLE 3
The Constant Terms Used in Our ILP Formulation

Constant	Definition
D	Number of disks
R	Number of requests
L	Number of RPM levels of disks
T	Time constraint
C_{sup}	Energy consumed to spin-up a disk
C_{sdown}	Energy consumed to spin-down a disk
T_{sup}	Time spent to spin-up a disk
T_{sdown}	Time spent to spin-down a disk
W	Delay to ensure a sufficiently long period for TPM
$Req_{d,r}(t, load)$	Access request r to disk d has the following parameters: Request is made at time t with the amount of I/O, $load$.
$RPM_l(rate, ae, ie, se)$	Each RPM level l has service rate $rate$, active energy ae, idle energy ie, and standby energy se
$RCCost_{l_1,l_2}$	Indicates the energy consumption when the RPM level is changed from l_1 to l_2

TABLE 4
The Variables Used in Our ILP Formulation

Constant	Definition
$DRPM_{d,l,r}$	Indicates whether disk d is running on RPM level l for the duration of the request r
$TPM_{d,r}$	Indicates whether disk d uses TPM for request r
$Len_{d,r}$	The duration of disk d servicing request r
$Start_{d,r}$	The start time of service for request r by disk d
$Finish_{d,r}$	The end time of service for request r by disk d
$DLen_{d,l,r}$	The duration of disk d running on RPM level l for request r
$TLen_{d,r}$	The duration of disk d using TPM for request r
$Inactive_{d,r}$	The duration of disk d not servicing any requests after r
$Idle_{d,r}$	The idle period of disk d after finishing request r
$IdleE_{d,r}$	The idle energy consumption of disk d for request r
$Standby_{d,r}$	The standby period of disk d after finishing request r
$RPMC_{d,l_1,l_2,r}$	The RPM level of disk d is changed from l_1 to l_2 just before r

useful in general since it can capture the scenarios with multiple disks, multiple CPUs, and multiple power modes. Note that our ILP-based formulation neither resolves the dependency between I/O requests nor finds the best ordering of I/O requests experienced by a given program execution since the disk trace of the restructured code already captures the dependencies among different disk requests (however, if explicit dependency information is given, our ILP formulation can be modified to account for that as well). Our ILP formulation instead determines the optimal disk power mode (for example, TPM or DRPM) and its value (for example, optimal RPM level) that maximize energy savings using a given disk trace. ILP provides a set of techniques that solve those optimization problems in which both the objective function and constraints are linear functions, and the solution variables are restricted to be integers. The 0-1 ILP (also known as ZILP) is an ILP problem in which each (solution) variable is restricted to be either 0 or 1 [24]. It is used in this paper for determining the RPM levels of disks and the times at which to switch them. Table 3 gives the constant terms used in our ILP formulation. We used *Xpress MP* [38], a publicly available tool to solve the resulting ILP problem.

Our objective is to find the RPM level of each disk during servicing the requests for minimum energy consumption. Based on a given number of disks and processors, we determine the RPM level of each disk using the disk access request pattern. We define 0-1 variables for each disk and for every request. By using these 0-1 variables, we determine whether TPM is used or not. Also, RPM levels can be captured by these variables.

Table 4 lists the variables used in our ILP formulation. We use 0-1 variables to specify the RPM level of each disk. Specifically, a disk can be in two different states: DRPM ($DRPM_{d,l,r}$) and TPM ($TPM_{d,r}$). Although the former uses a specific RPM level until the next request, the latter uses the maximum RPM level until servicing the current request and spins down until the next request if doing so reduces the energy consumption. We consider both TPM and DRPM in describing our set of ILP equation because we want to formulate it in a more generic form that captures various scenarios. This is because, depending on the disk idle period a disk experiences, the TPM might generate better energy savings than DRPM, whereas, in general, DRPM performs better for shorter idle periods. We also want to mention that our ILP formulation can be modified (if desired) to employ only TPM or DRPM. For example, in (3), we can omit the $TPM_{d,r}$ term and all associated terms (for example, T_{sup} and T_{sdown} in (13)) if we want to consider only DRPM. A disk may use different RPM levels for different requests. Since a disk must be in one of these two states, the following constraint must hold:

$$TPM_{d,r} + \sum_{i=1}^{L} DRPM_{d,i,r} = 1, \quad \forall d, r. \tag{3}$$

In the above equation, i iterates over the RPM levels. The duration of a service depends on the RPM level of the disk for the given request. For a given request, either TPM or DRPM can be used because we want to select any of them that gives the best result in our objective function, which is to minimize the total energy consumption. Consequently, we use $DLen$ and $TLen$ to distinguish between these two:

$$DLen_{d,l,r} = DRPM_{d,l,r} \times \left(\frac{Req_{d,r}(t, load)}{RPM_l(rate)}\right), \quad \forall (d, l, r). \tag{4}$$

In the above equation, if the disk is using DRPM

$$(DRPM_{d,l,r} = 1),$$

then the time it takes to process the request is found by dividing the request load ($Req_{d,r}(t, load)$) by the service rate ($RPM_l(rate)$). Similarly, we have

$$TLen_{d,r} = TPM_{d,r} \times \left(\frac{Req_{d,r}(t, load)}{RPM_{max}(rate)}\right), \quad \forall (d, r). \tag{5}$$

Since TPM uses the maximum RPM, the request load should be divided by the service rate of the maximum RPM. The final length is captured by

$$Len_{d,r} = TLen_{d,r} + \sum_{i=1}^{L} DLen_{d,i,r}, \quad \forall (d, r). \tag{6}$$

A new request is serviced only if the previous request is finished. We express this constraint as follows:

$$Start_{d,r} \geq Finish_{d,r-1}, \quad \forall(d,r) \quad r \geq 2. \tag{7}$$

A request can start only after it is actually requested:

$$Start_{d,r} \geq Req_{d,r}(t, load), \quad \forall(d,r). \tag{8}$$

Here, $Req_{d,r}(t, load)$ is the time of the request. A request finishes only after it is serviced, i.e., we have

$$Finish_{d,r} = Start_{d,r} + Len_{d,r}, \quad \forall(d,r). \tag{9}$$

A disk is inactive between the end of a request and the start of the next one, which can be expressed as follows:

$$Inactive_{d,r} = Start_{d,r} - Finish_{d,r-1}, \quad \forall(d,r) \quad r \geq 2. \tag{10}$$

For the first request of each disk $Inactive_{d,1} = Start_{d,1}$, since the disk is inactive until the first request is serviced. During the inactive period, a disk can be in one of the two modes, idle or standby:

$$Inactive_{d,r} = Idle_{d,r} + Standby_{d,r}, \quad \forall(d,r). \tag{11}$$

In DRPM, a disk cannot be in the standby mode; that is, the disk should be in the idle mode. To ensure this, we use the following constraint:

$$Standby_{d,r} \leq TPM_{d,r} \times MAXINT, \quad \forall(d,r). \tag{12}$$

If the disk is using DRPM, then $TPM_{d,r} = 0$. Consequently, $Standby_{d,r} \leq 0$. However, if the disk is using TPM, then $Standby_{d,r} \leq MAXINT$, which lets $Standby_{d,r}$ to have a suitable value. TPM can be used only if the idleness period is sufficiently enough, and we can capture this using the following expression:

$$\begin{aligned} Inactive_{d,r} \;\geq\; & TPM_{d,r} \times (T_{sup} + T_{sdown} + W) \\ & + Standby_{d,r}, \quad \forall(d,r). \end{aligned} \tag{13}$$

The disk needs to spin down (T_{sdown}) and spin back up (T_{sup}). In the above formulation, W is used as a delay to ensure a sufficiently long period for TPM. On the other hand, if DRPM is used, this constraint will not have any effect since it is already covered by (11). RPM change in a disk requires spin up/down, which, in turn, consumes certain energy. RPM changes are captured by the use of $RPMC_{d,l_1,l_2,r}$. This indicates that the RPM level of disk d is changed from l_1 to l_2 at a request r. Specifically, we have

$$\begin{aligned} RPMC_{d,l_1,l_2,r} &\geq DRPM_{d,l_2,r} + DRPM_{d,l_1,r-1} - 1, \\ RPMC_{d,l_1,max,r} &\geq TPM_{d,r} + DRPM_{d,l_1,r-1} - 1, \\ RPMC_{d,max,l_1,r} &\geq DRPM_{d,l_1,r} + TPM_{d,r-1} - 1, \\ \forall(d,l_1,l_2,r) &\quad \text{such that} \quad l_1 \neq l_2, \quad r \geq 2. \end{aligned} \tag{14}$$

The first constraint is used for the RPM changes within the DRPM. The second and third constraints, however, are used for a transition between TPM and DRPM. Here, max denotes the maximum RPM level since TPM uses the maximum RPM level.

Having specified the necessary constraints in our ILP formulation, we next give our objective function. In our disk energy model, there are four components of the total memory energy consumption:

- *Active*. The energy consumed when the disk is servicing a request.
- *Idle*. The energy consumed when the disk is running but not servicing any requests.
- *Standby*. The energy consumed when the disk is in standby mode.
- *Spin up/down*. The energy consumed to spin up/down a disk.

The active energy is composed of DRPM and TPM energies. Therefore, we can write

$$\begin{aligned} E_A = & \sum_{i=1}^{D}\sum_{j=1}^{L}\sum_{k=1}^{R} DLen_{i,j,k} \times RPM_j(ae) \\ & + \sum_{i=1}^{D}\sum_{j=1}^{R} TLen_{i,j} \times RPM_{max}(ae). \end{aligned} \tag{15}$$

The first term in the above expression is for DRPM, and the second one is for TPM. The idle energy consumption for each disk d and for each request r is captured by the following expression:

$$IdleE_{d,r} \geq Idle_{d,r} \times RPM_l(ie), \forall(d,l,r). \tag{16}$$

We can obtain the idle energy consumption by adding these $IdleE_{d,r}$ values:

$$E_I = \sum_{i=1}^{D}\sum_{j=1}^{R} IdleE_{i,j}. \tag{17}$$

In a similar fashion, we capture the standby energy as follows:

$$E_S = \sum_{i=1}^{D}\sum_{j=1}^{R} Standby_{i,j} \times RPM_{max}(se). \tag{18}$$

Spin-up/down energy is composed of two portions. The first portion is due to TPM going into standby mode, and the second one is due to the RPM level change:

$$\begin{aligned} E_P = & \sum_{i=1}^{D}\sum_{j=1}^{R} TPM_{i,j} \times \left(RCCost_{0,max} + RCCost_{max,0}\right) \\ & + \sum_{i=1}^{D}\sum_{j=1}^{L}\sum_{k=1,k\neq j}^{L}\sum_{l=1}^{R} RPMC_{i,j,k,l} \times RCCost_{j,k}. \end{aligned} \tag{19}$$

The first part corresponds to the spin-up/down energy due to the standby mode. Here, although max denotes the maximum RPM level, 0 denotes the standby mode. Each TPM will incur a spin up from standby mode to maximum RPM level and a spin down from maximum RPM level to standby mode. Similarly, each RPM change between DRPM and/or TPM will incur some energy consumption due to spin up/down. This is expressed in the second part of the above expression.

Based on these constraints, we can express the disk energy consumption (E) as follows:

$$E = E_A + E_I + E_S + E_P. \tag{20}$$

In this formulation, E_A, E_I, E_S, and E_P correspond to active energy, idle energy, standby energy, and spin-up/down energy, respectively. Based on this formulation, our

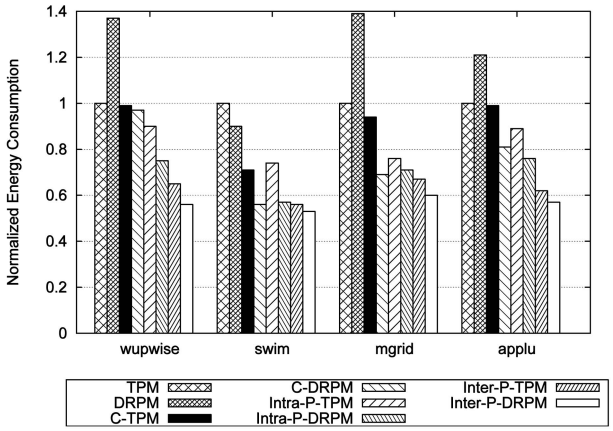


Fig. 11. Energy consumptions with different schemes.

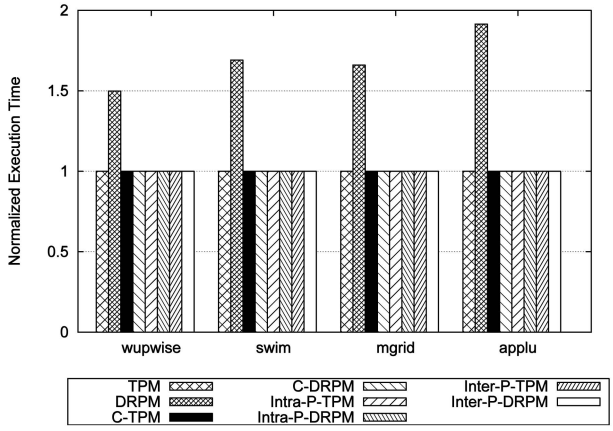


Fig. 12. Execution cycles with different schemes.

0-1 ILP problem can be defined as one of "minimizing E under constraints (3) through (19)."

7 Experimental Results

The graph in Fig. 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 Fig. 13. Moreover, 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 Fig. 3). In comparison, C-TPM brings about 9 percent 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 percent 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 percent and 43 percent savings in disk energy. In comparison, Intra-P-TPM and Intra-P-DRPM save approximately 18 percent and 30 percent 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 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 (that is, execution times/cycles) as well. The bar chart in Fig. 12 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 percent on the 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

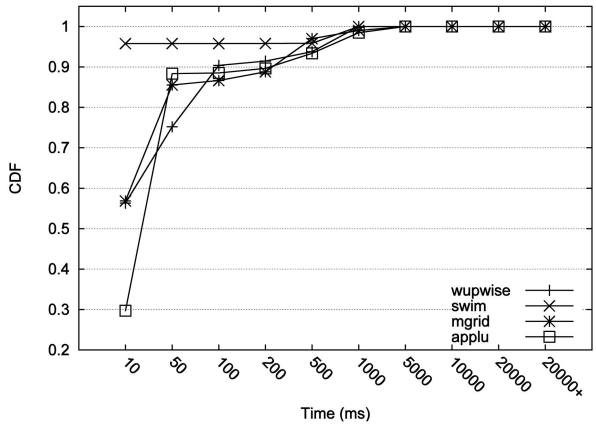


Fig. 13. Cumulative distribution function (CDF) curves for disk idle times. An (x, y) point on a curve indicates that y percent of the idle times has a duration of x (ms) or lower. As mentioned earlier, the minimum amount of idle time required to compensate the cost of spinning down the disk and up (under 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.

discussed earlier. We also see that all the compiler-directed schemes, namely, 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 the disk idle periods for the application domain we target. These results, along with those presented in Fig. 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

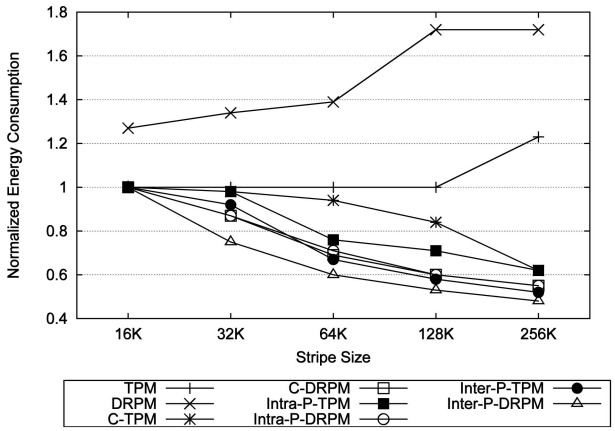


Fig. 14. Impact of stripe size on energy consumption.

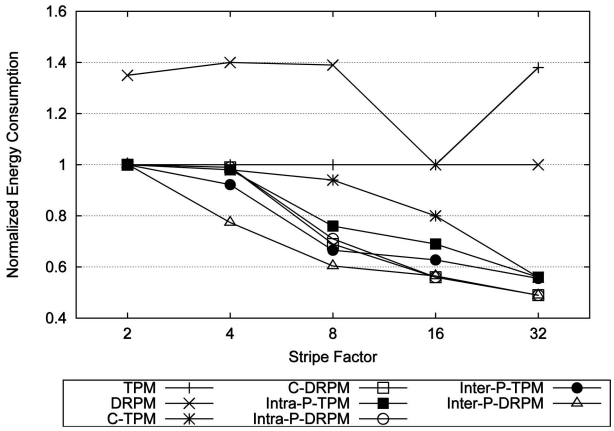


Fig. 15. Impact of stripe factor on energy consumption.

further performance data, mainly because, except for the DRPM scheme, none of the schemes evaluated causes any substantial increase in the original execution cycles. More specifically, except for DRPM, the average execution time increase was always less than 1 percent. 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.

Fig. 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 (16 Kbyte), disks do not experience much idleness. In fact, the disk idleness in this case becomes so small that even code restructuring cannot take much advantage of it. When the stripe size is increased, more disk requests can be serviced by a single stripe, that is, 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 Fig. 14 that Inter-P-DRPM generates the best savings with 256 Kbyte 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 eight. Fig. 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 Fig. 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 the impact of our compiler-based schemes are concerned.

The next parameter whose variation we study is the number of processors. Fig. 17 gives the normalized energy results with the 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 a large number of processors. This problem does not usually exist in the code-restructuring-based schemes since they cluster idle and active periods.

Our experimental results presented so far indicate that our scheme successfully restructures a given program code in such a way that the length of the disk idle time increased, which, in turn, allows more disk to be placed into the low

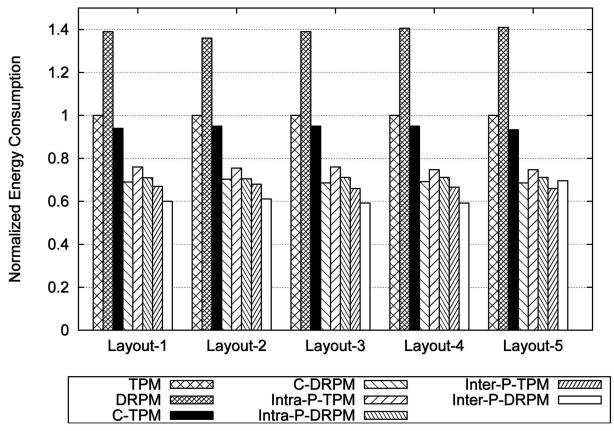


Fig. 16. Impact of starting disk on energy consumption.

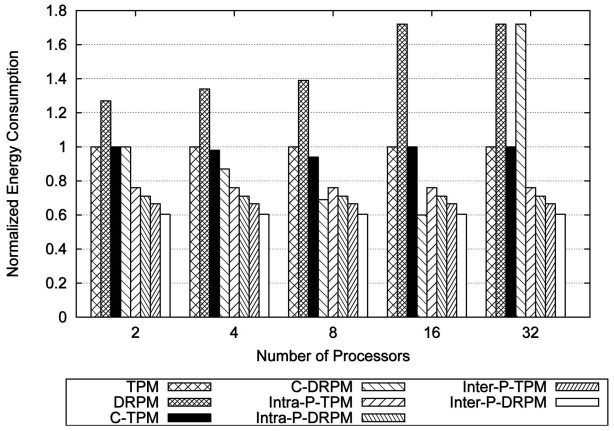


Fig. 17. Impact of processor count on energy consumption.

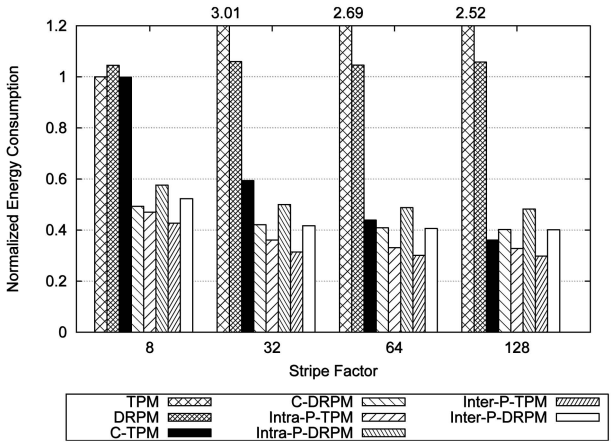


Fig. 19. Scalability of various schemes experimented.

power mode for a longer duration. In the next set of experiments, we would like to show how close the disk RPM level (either in TPM or DRPM) determined by our scheme comes to the optimal one by the ILP formulation. Recall that the problem we want to tackle using the ILP formulation described in Section 6 is to determine the optimal disk mode and value (for example, TPM versus DRPM or optimal RPM level in the case of DRPM) with given disk traces, which is generated by one of our code-restructuring schemes. The graph in Fig. 18 gives the normalized energy consumption values for the Inter-P-DRPM and ILP schemes with respect to the Base version explained in Section 6. We chose the Inter-P-DRPM scheme in this comparison because it gives the best energy savings among the schemes tested so far. Since it is infeasible to solve the ILP problem described in Section 6 with the whole disk traces, we obtained the results given in Fig. 18 using a small fraction of the total number of traces that is less than 5 percent but captures most representing access patterns of each benchmark. We can see from these results that the ILP solution gives slightly better energy savings than the Inter-P-DRPM scheme. Specifically, the difference between two schemes is only 3 percent on the average. This shows that our approach generates near-optimal solutions for all four benchmarks as compared to the optimal energy consumption achieved by the ILP solver. We also want to mention

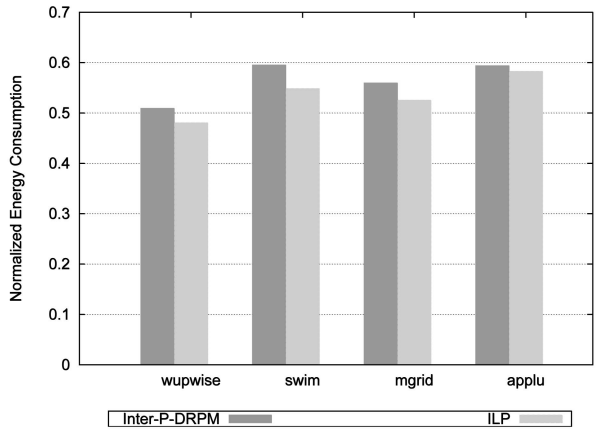


Fig. 18. Energy consumption with the Inter-P-DRPM- and ILP-based schemes.

that the ILP-based solution is not scalable. For our applications, even with limited number of trace data, it took more than 24 hours to generate optimal results. We present the ILP results here only to demonstrate that our approach is very effective in practice and generates near-optimal results.

In our next set of experiments, we conduct an experiment with up to 128 disks to see how the energy savings we achieved are affected with the larger number of disks. We also increased the problem size as we deal with the larger amount of data. Fig. 19 shows the normalized energy consumption results with the different number of disks, from 8 to 128. As we can see, in most cases, the compiler-directed schemes achieve better energy savings than the hardware-based schemes such as TPM and DRPM. One noticeable trend that can be observed is that the TPM scheme increases energy consumption dramatically with the larger number of disks. The main reason for this is that TPM works in a reactive manner. That is, even if the disk can have more chances to spin down due to the increase in disk idle time with the increased number of disks, it also needs to pay spin-up cost, which incurs both energy and performance penalties. Another observation we can make from these results is that, as the number of disks increases, the TPM-flavored compiler schemes exhibit better energy savings than the DRPM flavored ones. This is because spinning down to shutdown mode consumes much less energy than modulating the disk in a lower RPM, even if it is the lowest one.

Last, we want to show how our approach can be used with other techniques to achieve further energy savings. As shown in [32], optimizing disk layout alone can also bring a significant amount of energy savings. We present in Fig. 20 the normalized energy consumption for mgrid with 16 different schemes: eight original schemes we experimented so far and the remaining eight schemes are the ones that employ the OPT scheme, which determines energy-optimal disk layout for each array with minimal degradation on performance. As we can see from these results, the schemes with OPT generate from 6 to 20 percent additional energy savings over the original schemes, depending on the scheme in question. This result clearly shows that the energy savings from our approach can be increased further when it is used in conjunction with other energy-saving techniques targeting the disk subsystem.

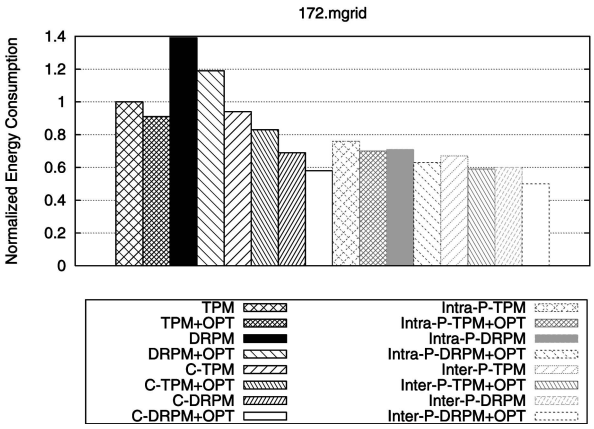


Fig. 20. Normalized energy consumption for our approach when used in conjunction with energy-efficient disk layout scheme.

8 Concluding Remarks

Since disk subsystems of parallel architectures are known to consume a large fraction of the overall power budget, they are an important optimization target. 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 are very promising and show that the proposed compiler-driven proactive approach to disk power management performs much better than existing hardware-based techniques. Furthermore, when compared with the results obtained from an ILP formulation, our approach can generate near-optimal energy savings.

Acknowledgments

A preliminary version of this paper appears in the *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)* [33]. This draft extends the PPoPP paper by providing an Integer Linear Programming (ILP) formulation of the problem to illustrate how close the results obtained by our approach are to the results obtained through the ILP solver. This work is supported in part by US National Science Foundation Grants 0444158, 0406340, and 0093082 and a grant from Gigascale Systems Research Center (GSRC).

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," PhD thesis, CSL-TR-97-719, Stanford Univ., 1997.

[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. Design Automation of Electronic Systems,* vol. 4, no. 4, pp. 351-375, 1999.

[4] J.S. Bucy et al., "The DiskSim Simulation Environment Version 3.0 Reference Manual," Technical Report CMU-CS-03-102, CMU, Jan. 2003.

[5] E.V. Carrera, E. Pinheiro, and R. Bianchini, "Conserving Disk Energy in Network Servers," *Proc. 17th Int'l Conf. Supercomputing,* pp. 86-97, June 2003.

[6] J.S. Chase, D.C. Anderson, P.N. Thackar, A.M. Vahdat, and R.P. Boyle, "Managing Energy and Server Resources in Hosting Centers," *Proc. 18th Symp. Operating Systems Principles,* pp. 103-116, Oct. 2001.

[7] J.S. Chase and R.P. Doyle, "Balance of Power: Energy Management for Server Clusters," *Proc. Eighth Workshop Hot Topics in Operating Systems,* p. 165, May 2001.

[8] X. Chen and L. Peh, "Leakage Power Modeling and Optimization in Interconnection Networks," *Proc. Int'l Symp. Low Power and Electronics Design,* pp. 90-95, Aug. 2003.

[9] F. Douglis, P. Krishnan, and B. Bershad, "Adaptive Disk Spin-Down Policies for Mobile Computers," *Proc. Second Symp. Mobile and Location-Independent Computing,* pp. 121-137, 1995.

[10] F. Douglis, P. Krishnan, and B. Marsh, "Thwarting the Power-Hungry Disk," *Proc. Usenix Winter Conf.,* pp. 292-306, 1994.

[11] E.N.M. Elnozahy, M. Kistler, and R. Rajamony, "Energy-Efficient Server Clusters," *Proc. Second Workshop Power Aware Computing Systems,* Feb. 2002.

[12] M. Elnozahy, M. Kistler, and R. Rajamony, "Energy Conservation Policies for Web Servers," *Proc. Fourth Usenix Symp. Internet Technologies and Systems,* Mar. 2003.

[13] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke, "DRPM: Dynamic Speed Control for Power Management in Server Class Disks," *Proc. Int'l Symp. Computer Architecture,* pp. 169-179, June 2003.

[14] S. Gurumurthi, J. Zhang, A. Sivasubramaniam, M. Kandemir, H. Franke, N. Vijaykrishnan, and M.J. Irwin, "Interplay of Energy and Performance for Disk Arrays Running Transaction Processing Workloads," *Proc. Int'l Symp. Performance Analysis of Systems and Software,* pp. 123-132, Mar. 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,* vol. 29, no. 12, pp. 84-89, Dec. 1996.

[16] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini, "Application Transformations for Energy and Performance-Aware Device Management," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 121-130, Sept. 2002.

[17] "Hitachi Power and Acoustic Management—Quietly Cool," white paper, Hitachi Global Storage Technologies, http://www.hitachigst.com/tech/techlib.nsf/productfamilies/White_Papers, Mar. 2004.

[18] "Adaptive Power Management for Mobile Hard Drives," technical report, IBM Storage Systems Division, Apr. 1999, http://www.almaden.ibm.com/almaden/pbwhitepaper.pdf.

[19] "Ultrastar 36Z15 Hard Disk Drive," IBM, http://www.hitachigst.com/hdd/ultra/ul36z15.htm, 2001.

[20] E.J. Kim, K.H. Yum, G. Link, M.K.N. Vijaykrishnan, M.J. Irwin, M. Yousif, and C.R. Das, "Energy Optimization Techniques in Cluster Interconnects," *Proc. Int'l Symp. Low Power Electronics and Design,* pp. 459-464, Aug. 2003.

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

[22] K. Li, R. Kumpf, P. Horton, and T. Anderson, "A Quantitative Analysis of Disk Drive Power Management in Portable Computers," *Proc. Usenix Winter Conf.,* pp. 279-292, 1994.

[23] S.S. Muchnick, *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[24] G.L. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization.* Wiley-Interscience, 1988.

[25] A.E. Papathanasiou and M.L. Scott, "Energy Efficient Prefetching and Caching," *Proc. Usenix Ann. Technical Conf.,* pp. 255-268, 2004.

[26] M. Pedram, "Power Optimization and Management in Embedded Systems," *Proc. Conf. Asia South Pacific Design Automation,* pp. 239-244, 2001.

[27] T. Pering, T. Burd, and R. Brodersen, "The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms," *Proc. Int'l Symp. Low Power Electronics and Design,* Aug. 1998.

[28] P. Pillai and K.G. Shin, "Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems," *Proc. 18th ACM Symp. Operating Systems Principles,* pp. 89-102, Aug. 2001.

[29] W. Pugh, "A Practical Algorithm for Exact Array Dependency Analysis," *Comm. ACM*, vol. 35, no. 8, pp. 102-114, Aug. 1992.
[30] R.B. Ross, P.H. Carns, W.B. Ligon III, and R. Latham, "Using the Parallel Virtual File System," http://www.parl.clemson.edu/pvfs/user-guide.html, July 2002.
[31] T. Simunic, L. Benini, and G.D. Micheli, "Energy-Efficient Design of Battery-Powered Embedded Systems," *Proc. Int'l Symp. Low Power Electronics and Design*, pp. 212-217, Aug. 1999.
[32] S.W. Son, G. Chen, and M. Kandemir, "Disk Layout Optimization for Reducing Energy Consumption," *Proc. 19th ACM Int'l Conf. Supercomputing*, pp. 274-283, June 2005.
[33] S.W. Son, G. Chen, M. Kandemir, and A. Choudhary, "Exposing Disk Layout to Compiler for Reducing Energy Consumption of Parallel Disk Based Systems," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 174-185, June 2005.
[34] "CFP 2000," SPEC, http://www.specbench.org/cpu2000/CFP2000/, 2000.
[35] "Analyzing Program Performance with Sun WorkShop," Sun Microsystems, http://doc-pdg.sun.com/806-7989/807-7989.pdf, May 2000.
[36] M. Weiser, A. Demers, B. Welch, and S. Shenker, "Scheduling for Reduced CPU Energy," *Proc. Symp. Operating System Design and Implementation*, pp. 13-23, Nov. 1994.
[37] M. Wolfe, *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
[38] Xpress-MP, http://www.dashoptimization.com/pdf/Mosel1.pdf, May 2002.
[39] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes, "Hibernator: Helping Disk Arrays Sleep through the Winter," *Proc. 20th ACM Symp. Operating Systems Principles*, Oct. 2005.
[40] Q. Zhu, F.M. David, C.F. Devaraj, Z. Li, Y. Zhou, and P. Cao, "Reducing Energy Consumption of Disk Storage Using Power-Aware Cache Management," *Proc. 10th Int'l Conf. High-Performance Computer Architecture*, pp. 118-129, 2004.
[41] Q. Zhu, A. Shankar, and Y. Zhou, "PB-LRU: A Self-Tuning Power Aware Storage Cache Replacement Algorithm for Conserving Disk Energy," *Proc. 18th Ann. Int'l Conf. Supercomputing*, pp. 79-88, 2004.

Seung Woo Son received the BE and ME degrees in computer engineering from Yeungnam University, Korea, in 1995 and 1997, respectively. He is a PhD candidate in the Department of Computer Science and Engineering, Pennsylvania State University. From 1997 to 2003, he was a research engineer at the Electronics and Telecommunications Research Institute in Korea. His research interests include power/energy optimization for storage systems and embedded software. He is a student member of the IEEE and the ACM.

Guangyu Chen received the PhD degree in computer science and engineering from the Pennsylvania State University in 2006. He is a software engineer at Microsoft. His research interests include software technologies for embedded systems, resource-constrained Java virtual machines, and on-chip networks for multicore processors. He is a student member of the IEEE.

Ozcan Ozturk received the BSc degree in computer engineering from Bogazici University, Istanbul, in 2000 and the MS degree in computer engineering from the University of Florida, Gainesville, in 2002. He is currently pursuing the PhD degree in computer science and engineering at the Pennsylvania State University. His research interests include on-chip multiprocessing, power-aware architectures, and compiler optimizations. He is a student member of the IEEE.

Mahmut Kandemir received the BSc and MSc degrees in control and computer engineering from Istanbul Technical University, Istanbul, in 1988 and 1992, respectively, and the PhD degree in electrical engineering and computer science from Syracuse University, Syracuse, New York, in 1999. He is an associate professor in the Department of Computer Science and Engineering, Pennsylvania State University. His main research interests include optimizing compilers, I/O intensive applications, and power-aware computing. His research is currently funded by the US National Science Foundation, DARPA, and SRC. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Alok Choudhary received the MS degree from the University of Massachusetts, Amherst, in 1986 and the PhD degree in electrical and computer engineering from the University of Illinois, Urbana-Champaign, in 1989. He is a professor in the Electrical and Computer Engineering Department, Kellogg School of Management, Northwestern University. He joined Northwestern in 1996. He is also the chair of the Computer Engineering and Systems Division. Prior to that, he was a faculty member of the ECE Department at Syracuse University. His research interests are in databases and data warehouses, storage systems, super computing and parallel computing, embedded systems, computer architecture, e-commerce and Web-based systems, system software and algorithms, data mining, marketing and analytical marketing, customer relationship management, business intelligence, and information security. He is also interested in high-level synthesis on Systems-on-Chip applying compilation principles to the synthesis process. He has published more than 300 papers in various journals. He has also written a book and several book chapters. His research has been sponsored by (past and present) DARPA, the US National Science Foundation (NSF), NASA, AFOSR, ONR, DoE, Intel, IBM, and TI. He is an area editor of the *Journal of Parallel and Distributed Computing*, and served as an associate editor of the *IEEE Transactions on Parallel and Distributed Systems*. He has served as a guest editor of *Computer*. He is the founder and director of the Center for Ultra-scale Computing and Information Security (CUCIS). He is a fellow of the IEEE. He cofounded Accelchip, a developer of electronic design automation tools and services and served as its vice president for research and technology from 2000 to 2002. He also cofounded Nimkathana, a company that provided consulting services in the area of software systems, high-performance computing, cluster systems, data mining, databases, data warehousing, and other related topics. He has served or serves on technical advisory board of several companies. He is also on the board of directors of the C3Research and is the director of the CRM practice of C3Research. He has consulted with various small and large (including Fortune 100) companies on various topics in the areas of technology, management, and marketing. He served as the conference cochair for the International Conference on Parallel Processing, and as the chair of the International Workshop on I/O Systems in Parallel and Distributed Systems. He has served on the program committees of more than 50 conferences. He received the NSF's Young Investigator Award in 1993 (1993-1999). He has also received an IEEE Engineering Foundation award, an Intel research council award (1993-1997, 2003-2005), and an IBM Faculty Development award.