Runtime System Support for Software-Guided Disk Power Management

Seung Woo Son, Mahmut Kandemir

Department of Computer Science and Engineering
The Pennsylvania State University, University Park, PA 16802, USA
{sson,kandemir}@cse.psu.edu

***Abstract*— Disk subsystem is known to be a major contributor to the overall power budget of large-scale parallel systems. Most scientific applications today rely heavily on disk I/O for out-of-core computations, checkpointing, and visualization of data. To reduce excess energy consumption on disk system, prior studies proposed several hardware or OS-based disk power management schemes. While such schemes have been known to be effective in certain cases, they might miss opportunities for better energy savings due to their reactive nature. While compiler based schemes can make more accurate decisions on a given application by extracting disk access patterns statically, the lack of runtime information on the status of shared disks may lead to wrong decisions when multiple applications exercise the same set of disks concurrently. In this paper, we propose a runtime system based approach that provides more effective disk power management. In our scheme, the compiler provides crucial information on the future disk access patterns and preferred disk speeds from the perspective of individual applications, and a runtime system uses this information along with current state of the shared disks to make decisions that are agreeable to all applications. We implemented our runtime system support within PVFS2, a parallel file system. Our experimental results with four I/O-intensive scientific applications indicate large energy savings: 19.4% and 39.9% over the previously-proposed pure software and pure hardware based schemes, respectively. We further show in this paper that our scheme can achieve consistent energy savings with a varying number and mix of applications and different disk layouts of data.**

I. Introduction

Reducing power consumption of disk systems can be very important for high-performance architectures where disks are responsible for a large fraction of overall power budget [1], [2], [3], [4]. As system sizes and capabilities approach the petascale range, one can expect scientific applications to be even more data hungry in the future, which means more frequent use of the disk storage systems. Prior research studied disk power reduction techniques from both the hardware and software perspectives. Hardware related work in this area includes spinning down idle disks [5], [6] and employing multi-speed disks [1], [7]. Software work on the other hand focuses on file system based approaches such as [8] and compiler related studies [9], [10], [11].

Compiler based approaches to disk power management generally operate under two important assumptions: i) disk layouts of data sets are available to the compiler so that it can make power management decisions at the source code level using the data-to-disk mapping, and ii) the application being optimized for disk power reduction is the sole customer of the disk system when it is scheduled to execute. While several file systems and runtime libraries today already provide interfaces to query/control disk layout of data (addressing the first assumption above), the second assumption can only be satisfied in environments which do not execute two or more applications that concurrently exercise the same disk system.

When multiple applications use the same set of disks concurrently, a *runtime system based approach* may be more suitable for reducing disk power consumption. Motivated by this observation, in this work, we investigate a runtime system centric disk power reduction scheme and quantify its impact on a set of scientific applications that process disk-resident data. Our approach is a cooperative one that uses help from both architecture and compiler. The role of the architecture in our work is to provide multi-speed rotation capability for disks and the role of the compiler is to analyze the application source code and extract the disk access patterns using the data-to-disk mapping exposed to it. The proposed runtime system receives, as *hints*, the preferred disk speeds from individual applications and, considering all hints, decides the best rotational speed for each disk in the system such that overall power consumption is reduced without affecting performance.

In a sense, this approach combines the best characteristics of the pure hardware based and the pure compiler based schemes. Like the hardware based power management schemes, it can operate under multiple application executions (which is something that cannot be done using the compiler based schemes alone). In addition, like the pure compiler based schemes, it is proactive; that is, it can select the best disk speeds without requiring an (disk usage) observation period, which is typically required by the hardware based approaches to disk power reduction.

We implemented our runtime system support within PVFS2, a parallel file system [12] and performed experiments with four scientific applications that process disk-resident data sets. Our experimental results are very promising: We achieve, under the default values of our simulation parameters, 39.9% and 19.4% savings in disk power consumption, over the pure hardware based and the pure compiler based schemes, respectively. We further show in this paper that our scheme can achieve consistent energy savings with a varying number and mix of applications and different disk layouts of data.

The remainder of this paper is structured as follows. A discussion of the related work on disk power reduction is given in Section II. Sections III and IV describe, respectively, the architectural support and compiler support required by

our approach. Section V presents the details of our runtime system for software-based disk power management. Section VI gives our evaluation methodology and experimental results collected using several I/O-intensive scientific applications. Finally, Section VII concludes the paper with the summary of our major findings and briefly discusses the planned future work.

II. Discussion of Related Work

A simple mechanism for reducing disk power consumption is to spin down a disk when it is idle for a certain period of time. While a disk in such a spin-down mode consumes much less energy than disks in active mode, it must be spun up in order to service any upcoming requests, which may incur significant energy and performance penalties. Many studies considered techniques such as spinning down idle disks by using a fixed threshold [6], i.e., the amount of time to wait before spinning down a disk, or by estimating the threshold value adaptively [5]. A recent study considered program counter based techniques for predicting upcoming disk accesses [13]. While these approaches are certainly effective when targeting the laptop/desktop workloads where the applications typically exhibit long idle periods, this is rarely the case with I/O-intensive scientific workloads. To address this problem, multi-speed disk models are proposed [7], [1], in which a disk can service I/O requests even when it is spinning at lower speeds. Such multi-speed disks (e.g., those from [14]) can exploit short idle periods very well by changing disk speed dynamically, and this can reduce power consumption (since power consumption is a function of disk speed [7]). In this paper, we assume that this type of multi-speed disk system is available to our runtime system based scheme.

Many researches proposed schemes to utilize low-power modes provided by disks, either spinning down or rotating disks at lower speeds. Observing that conventional OS caching and prefetching techniques can hardly produce any long idle periods, Zhu et al [15] and Papathanasiou et al [16] investigated energy-efficient caching and prefetching strategies. Their idea is to generate burst disk access patterns, which can be preferable to extending disk idle times, by delaying or prefetching/caching disk blocks selectively based on disk speeds. Zhu et al [15] also studied power-aware cache replacement schemes, called PA-LRU and PB-LRU, in the context of large storage systems, in which several gigabytes of aggregated caches can be redesigned to maintain file cache blocks from the disks which are frequently accessed, so that the remaining disks can remain in the low-power modes for longer periods of time.

As disk arrays are popular building-blocks for constructing large-scale storage systems, several studies focused on energy-efficient disk arrays in an attempt to trade off availability against disk energy by spinning down over-provisioned disk drives depending on the workload variation. These techniques include EERAID [17] and PARAID [18]. In a recent study, Zhu et al [19] proposed a holistic technique, called Hibernator, that combines disk block reorganization with two other techniques, namely, dynamic disk speed setting and multi-tier data layout. In order not to decrease disk reliability due to frequent disk speed modulations, Hibernator adjusts disk speeds at a coarse granularity. It also keeps track of the average response time at runtime, and restores the speeds of all disks to the highest one when the specified response time guarantee is at risk. More recently, Cai et al [20] discussed a power management scheme that considers reducing memory and disk energy simultaneously under certain performance constraints.

Another category of related studies investigated approaches that involve transforming application code to increase idle periods and informing the OS of the future disk idle periods. For example, Heath et al [9] studied such a technique in the context of laptop disks that can generate I/O burstiness in laptop applications. In comparison, Son et al [11], [10] proposed compiler-based code transformation techniques to conserve disk energy. Focusing on scientific applications with regular data access patterns, they first studied a compiler technique that inserts explicit disk power management calls based on the extracted information about the future disk access patterns [11], [10]. Explicit power management calls inserted by the compiler can potentially eliminate the performance penalty that would normally be incurred in reactive disk power management schemes. These studies assume that the disk system is exercised by a single application at a time. Such pure compiler-based techniques do not easily extend to scenarios where multiple applications exercise the same set of disks at the same time.

The last category of studies we discuss here investigate data reorganization schemes that cluster disk accesses onto a subset of disks, thereby increasing the chances for the remaining disks to remain in the low-power modes longer. MAID (Massive Arrays of Idle Disks) [21] uses a small number of cache disks to keep recently accessed blocks. As a consequence, it can potentially reduce the number of spin-ups for the remaining disks. PDC (Popular Data Concentration) [8], on the other hand, dynamically migrates the most frequently used files to a subset of the disks in the array. This scheme exploits the fact that the workloads in network servers are heavily skewed towards a small set of files. Since MAID and PDC migrate files depending on workload changes, they can incur performance degradation due to skewed disk I/O load.

Our approach in this paper is different from the pure hardware or OS based disk power management schemes as it utilizes compiler-guided decision on disk power management. It is also different from the prior compiler-based studies as it employs a runtime system that uses compiler-extracted information and the current status of the shared disks to select a speed for each disk that would be agreeable to all applications, instead of solely relying on the compiler-guided decision. In contrast to the previous compiler-based studies that target single-application-only scenarios such as [9], [10], and [11], our approach can handle multiple applications running in parallel and accessing the same shared storage system. Finally, as opposed to most of the prior studies that focus on array-based applications with disk-resident data, we consider MPI-IO based parallel scientific applications where file I/O is performed through explicit MPI-IO calls.

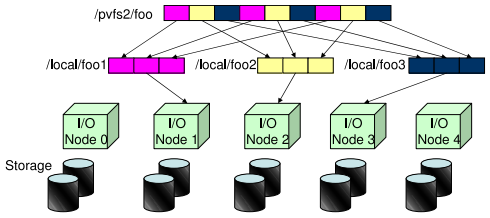


Fig. 1. Striping a file across multiple disks.

III. Architectural Support

The storage and file system architecture considered in this paper is depicted in Figure 1. Most parallel file systems available today either commercially or for research purposes provide high-performance I/O by *striping* (i.e., dividing a file into blocks called stripes and distributing these blocks in a round-robin fashion across multiple disks), as illustrated in Figure 1, to support parallel disk accesses for I/O-intensive applications that manipulate disk-resident, multidimensional arrays. In such a system, the disk access pattern of an application is strongly influenced by the striping parameters. In Figure 1, the data file, foo, is striped across three I/O nodes, starting from node 1. At each I/O node, a stripe assigned to that I/O node can be further striped at the RAID level (depending on the RAID level [22] adopted) for performance and reliability purposes. Therefore, in many parallel file systems, a given file is typically striped at two different layers, i.e., hardware layer through RAID and software layer through file striping. While the hardware level striping is hidden from the compiler and the operating system, the file system level striping can be exposed to the software. Furthermore, the file level striping can even be controlled through the APIs provided by modern file systems. For example, PVFS2 [12] provides an MPI hint mechanism that allows application programmers to change the default stripe distribution across the available I/O nodes. Other parallel file systems such as PFS [23] and GPFS [24] also provide similar facilities.

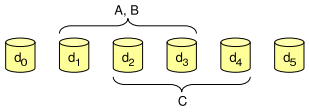
Since our goal in this work is to utilize the access pattern information extracted by the compiler analysis to perform disk power management, we focus on the I/O node layer. That is, our disk speed-setting decisions are made at an I/O node granularity. Therefore, the effect of changing the rotational speed on a specific I/O node simply means changing the rotational speeds of all the disks controlled by that I/O node. For the ease of discussion, however, we use the term "disk" instead of "I/O node" in the rest of the paper.

IV. Compiler Support

Our focus is on MPI-IO based parallel scientific applications. MPI-IO, which is the I/O part of the MPI-2 standard, is an interface that supports parallel I/O operations and optimizations [25], [26]. Our target applications access file-based data frequently for different purposes, e.g., reading and writing large data sets for out-of-core computations, storing the results of intermediate computations for long-running applications, and reading back data stored for checkpointing or visualization. One key characteristic of these applications is that, through automated code analysis, an optimizing compiler can identify logical file addresses along with the request size

```
/* Loop on horizontal slices */
for ih=0, R, 1 {
 /* Read next block of matrix A */
 MPI_File_read_at_all (fh_A, ···);
 /* Loop on vertical slices */
 for iv=1, R, 1 {
  /* Read next block of matrix B */
  MPI_File_read_at_all (fh_B, ···);
  /* Compute block product */
  for i=0, Nb, 1
   for j=0, Nb, 1
    for k=0, Nb, 1
     C[i][j] += A[i][k] * B[k][j];
  /* Write block of C */
  MPI_File_write_at_all (fh_C, ···);
 }
}
```

(a) Original code fragment.



(b) Disk layout for each array.

```
/* Loop on horizontal slices */
for ih=0, R, 1 {
 /* Read next block of matrix A */
 MPI_File_read_at_all (fh_A, ···);
 /* Loop on vertical slices */
 for iv=1, R, 1 {
  /* Read next block of matrix B */
  MPI_File_read_at_all (fh_B, ···);
  /* set the disk speed to 6K RPM */
  set_speed (011100, 6000, 0);
  /* Compute block product */
  for i=0, 3Nb/4, 1
   for (j=0, Nb, 1
    for k=0, Nb, 1
     C[i][j] += A[i][k] * B[k][j];
  /* set the disk speed to 15K RPM */
  set_speed (011110, 15000, 1);
  for i=3Nb/4+1, Nb, 1
   for j=0, Nb, Nb/4
    for k=0, Nb, 1
     C[i][j] += A[i][k] * B[k][j];
  /* Write block of C */
  MPI_File_write_at_all (fh_C, ···);
 }
}
```

(c) Transformed code fragment.

Fig. 2. Out-of-core matrix-matrix multiplication nest written using the MPI-IO calls. Each data slice (tile) is of size $Nb \times Nb$ array elements, and all three matrices, A, B, and C, have $R \times R$ number of slices, which corresponds to the number of processors.

of file blocks. To explain this better, let us consider the out-of-core blocked matrix multiplication nest shown in Figure 2(a). In this example, each processor accesses all three disk-resident arrays, A, B, and C, and opens sub-files that correspond to one row-block from A, and one column-block from B, and their intersection from C. It then reads data from A and B, carries out the corresponding partial multiplication in memory, and stores the results in C. Note that, the code given in Figure 2(a) will be executed using multiple processors and each processor performs both the I/O and computation assigned to it. As we can see from this code, it involves a substantial amount of disk I/O followed by some computation associated with it. Consequently, one can conceivably implement *proactive* disk power management by inserting explicit disk power management calls before and after the I/O phases. Since many scientific applications exhibit this type of behavior, we use this code fragment, as our example, to describe how a compiler can predict disk access patterns and modify the source code to insert explicit power management calls.

One of the important requirements in order to use a compiler in determining the most suitable disk speeds for upcoming I/O phases is to predict how disks are going to be accessed at a high level (source code level). We use the term *disk access pattern* in this paper to refer to the high-level information on the order in which the disks are accessed by a given application code. This pattern is crucial as it determines the durations of both active and idle periods for each individual disk in a disk system. In this work, we determine disk access patterns exhibited by each MPI-IO call in the code. To obtain this information, the compiler needs the file handle, file offset, and request size for each MPI-IO call (i.e., the parameters to the call), and the disk layout for each file. Since the file offset and request block size depend typically on the processor rank and loop bounds of the program, the compiler can obtain these

values by analyzing the application source code statically. Combining this information with the disk layout information exposed to it, the compiler can then understand how the disks will be exercised by a given MPI-IO call.

We next discuss how the disk layout abstraction discussed in Section III can be used to determine the disk being accessed by each I/O phase. As mentioned earlier, file striping is a technique that divides a large chunk of data into small data blocks (stripe) and stores these blocks on separate disks in a round-robin fashion (as depicted in Figure 1). This permits multiple processes to access different portions of the data concurrently without much disk contention. In addition to providing file striping, many parallel file systems today also provide a hint mechanism through which one can specify user-defined or manual file striping, as will be explained below. In this work, we represent the disk layout of a file using the following triplet:

(Base, Striping_Factor, Striping_Unit),

where Base is the first disk from which the file gets striped, Striping_Factor is the number of disks used for striping, and Striping_Unit (or stripe size) is the length of each stripe. In the PVFS2 file system [12], one can change the default striping parameters using the following MPI-IO hints that are supported: `CREATE_SET_DATAFILE_NODES` (enumeration of disk ID being used for striping), `Striping Factor`, and `Stripe Size`. Then, the striping information defined by the MPI_Info structure is passed to to MPI_File_open() calls. When creating a file within the application, this MPI hint information can be made available to the compiler as well. As explained above, the compiler uses this information in conjunction with file offset and request block size to determine the disk access pattern. If the file has already been created on the disk system, we can also obtain its layout by querying MPI_Info associated with the file in question. The obtained MPI_Info, which indicates the disk layout of the file, can then be passed to the compiler.

To demonstrate how to extract the disk access pattern using the code fragment shown in Figure 2(a), let us assume the disk layout in Figure 2(b). Assuming the same stripe size of 64Kbytes for all file stripes, the disk layouts for arrays A, B, C in this example are (1, 3, 64), (1, 3, 64), and (2, 3, 64), respectively. Let us further assume, for illustrative purposes, that the request size of each MPI_Read/MPI_Write call is of three stripe units, i.e., 64×3 Kbytes. Using the data access patterns and the file layout associated with them, the compiler can infer that, four disks (d_1, d_2, d_3, and d_4) must be run at the highest disk speed available in order not to degrade the I/O response time. On the other hand, since d_0 and d_5 are determined not to be used while executing the code fragment given in Figure 2(a), we can use for them the lowest disk speed available. Since the determined disk access patterns capture both the idle and active periods for each disk and their durations, we can insert (in the code) explicit power management calls. We also want to mention at this point that, in our approach, a power management call inserted by the compiler is actually a *hint* to the runtime system. That is, the runtime system does *not* need to obey all the speed-setting calls passed to it. Rather, it uses them to make a globally acceptable decisions regarding disk speeds. The details of how our runtime system makes that decision based on the compiler-inserted calls will be explained in Section V.

As mentioned above, the last part of our compiler support is to insert power management calls (set_speed() calls in our case) in the code. As an example, let us consider the code fragment in Figure 2(a) once again. Let us assume, for the sake of illustration, that a disk can spin at four different speeds, namely 6000, 9000, 12000, and 15000 RPM. In the previous step, we determine that, after reading the block of arrays A and B, the speeds of the disks just accessed (d_1, d_2, and d_3 in this example) can be set to the lowest speed available (6000 RPM). After the computation phase, array C must be written back to the disks. Consequently, the timing for restoring the disk speed to its maximum, 15000 RPM, is very important in order not to incur energy or performance overheads. More specifically, if the speed-setting calls are issued too early, the disk will be placed into the highest speed earlier than necessary. On the other hand, if they are issued too late, this can degrade performance. Since most scientific applications are built using nested loops, we use the number of loop iterations as our unit for deciding the point to insert the explicit disk speed-setting call, which is calculated as:

$$\lceil \frac{T_s}{T_b} \rceil, \tag{1}$$

where T_s is the cycles required to set the disk speed given as a parameter to set_speed() call, and T_b is the shortest possible execution latency (in cycles) through the loop body. Once the insertion point for the disk speed-setting call is determined, we use loop splitting [27] to make explicit the point where the call is to be inserted. The format of our disk speed-setting call is as follows:

set_speed (T, S_i, F),

where T is a tag that consists of D bits, where D is the number of disks in disk system. The bit in the jth position of T ($0 \leq j \leq D-1$) is set to 1 if the MPI-IO call accesses the jth disk. S_i ($1 \leq i \leq m$) is the ith speed level, and the last argument to the set_speed() call, F, is a flag which is set to 1 when the requested speed setting is for the upcoming I/O phase. Otherwise, F is set to 0 and this means that the disk will not be used for a certain period of time. Going back to our example code fragment in Figure 2(a), let us assume that the number of loop iterations required to change the disk speed from 6000 RPM to 15000 RPM is one fourth of the total number of iterations of the outermost loop, i. Therefore, the compiler splits the original loop nest into two parts: one with the first 3/4rd of iterations and one with the remaining 1/4th of iterations, and inserts the set_speed (011110, 15000, 1) call between these two newly-generated loop nests, as shown in Figure 2(c).

Our compiler algorithm for determining disk access patterns and identifying the desirable disk speed for each I/O call is given in Figure 3. This algorithm takes an input program along with available disk speeds and inserts explicit set_speed calls (hints) into the program. The first step of our compiler

INPUT:
Input program, $\mathcal{P}$;
Available disk speeds, $DS = (S_1, S_2, \ldots, S_m)$;
OUTPUT:
Transformed program, $\mathcal{P}'$;

DL := disk layout (base, striping_factor, striping_unit);
BV := bitvector indicating the disks being accessed;
T := tag consisting of D bits, where D is the number of disks;
T_s := the time required to change the speed of disks from x to y;

/* Step 1: Determine disk access patterns */
for each I/O call $\in \mathcal{P}$ {
  determine the I/O call parameters, i.e., file_id, file_offset, and size;
  query DL of file_id;
  if (size $\leq$ striping_unit) { /* small request size */
    i = (base + (file_offset % striping_unit) % D ;
    set ith bit to 1 in T;
  }
  else { /* large request size */
    for (j = base; j < base+striping_factor; j=j+1)
      set jth bit to 1 in T;
  }
}

/* Step 2: Determine the disk speed and splitting point */
for each loop nest, $\mathcal{L} \in \mathcal{P}$ {
  T_b = number of cycles required for executing the body of loop nest $\mathcal{L}$;
  for each available disk speed, $S_x \in DL${
    calculate T_s when disk speed is S_x;
    /* determine the loop iterations for hiding T_s */
    $d_x = \lceil \frac{T_s}{T_b} \rceil$;
    if ($2 \times d_x >$ total number of iterations $\in \mathcal{L}$) {
      select S_x as the preferable disk speed;
      select loop index l_x, where iterations of $l_x > d_x$;
    }
  }
}

/* Step 3: Insert set_speed() call hint to the code */
for $\mathcal{L} \in \mathcal{P}$ to be splitted {
  split $\mathcal{L}$ using d_x;
  emit set_speed (T, S_m, 1) at the split point;
  emit set_speed (T, S_x, 0) after the I/O call;
}

Fig. 3. Our compiler algorithm, in pseudo code, that determines the disk access patterns and finds the most desirable disk speeds for the disks accessed by each MPI-IO call.

algorithm is to determine the disk access patterns for each I/O call in the input program. As a result of the first step, the compiler determines the T parameters, which capture disk access patterns for each I/O call. The second step of our algorithm involves analyzing the loop nest that contains I/O calls and finding the desirable disk speed for each disk accessed by each I/O call. Based on the determined disk speed, the compiler then identifies the splitting point in the original loop nest. The last step of the algorithm is to transform the loop and insert the set_speed() calls in the determined points in the code.

Before moving to the next section, we would like to mention that the decisions made by the compiler explained so far are only valid from the perspective of a single application. Consequently, they may not be suitable when other (concurrently executing) applications that use the same set of disks are considered. Our proposed runtime support, which is explained next, is designed to address this problem. We also need to mention however that the compiler-determined preferable speeds are still very important. This is because if the optimized application happens to be the sole client of a particular disk, the runtime system will just use that preferable speed (indicated by the compiler) for the disk. Even if this application is not the only client of a disk, the runtime system still uses this hint in deciding the disk speed.

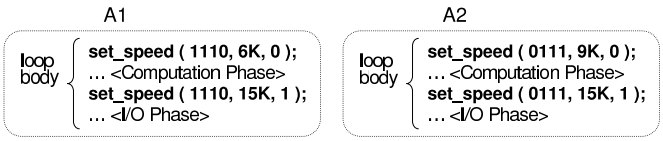


Fig. 4. Sketches of two example applications with the compiler-inserted disk power management calls (hints).

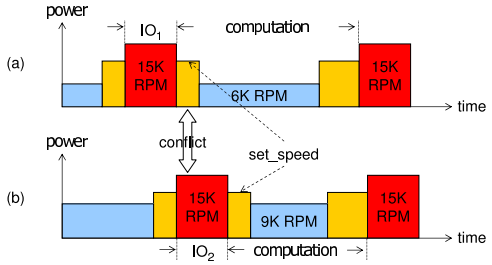


Fig. 5. Disk power state diagram for the applications in Figure 4. (a) A1 and (b) A2. The disk is assumed to have four discrete speeds; 6K, 9K, 12K, and 15K.

V. Runtime System Support

A. Functionality

We first address the problem of executing multiple applications concurrently, each of which is transformed to insert explicit disk speed-setting calls, as explained in the previous section. Figure 4 shows the sketches of two applications, A1 and A2, that are modified by the compiler (as explained in Section IV). In this execution scenario, it is assumed that A1 is assumed to have more idle periods than A2. Hence, the speed of the disks accessed by A1 is set to 6000 RPM by the compiler, whereas that of the disks accessed by A2 is set to 9000 RPM. Before the I/O phase of each application starts, the disks are set to the maximum speed, 15000. As explained in the previous section, the second set_speed() call of each application is issued early enough to hide the time required to restore the speed of disks to the maximum value. Figures 5(a) and (b) depict the disk power state diagram for A1 and A2, respectively, when A1 is started to execute followed by A2. Both A1 and A2 issue three set_speed() calls (hints) during their execution periods. While two of these calls do not have any impact on A2's performance and energy behavior, the second call issued by A1 has a substantial impact on the first I/O phase of A2, denoted using IO_2 in Figure 5(b). This is because the decision on disk speed made at the time when A1 issues set_speed (1110, 6000, 0) is lower than the one required for A2, which is 15000 RPM. In order not to incur a negative impact on the performance of A2, we need to set the speed of the shared disk to 15000 RPM, the higher of the conflicting speeds.

Figure 6 illustrates how our runtime system support handles the software-directed disk power management calls inserted by the compiler. To make a decision that would be agreeable to all concurrently-running applications, our runtime system needs to intercept each disk power management call inserted

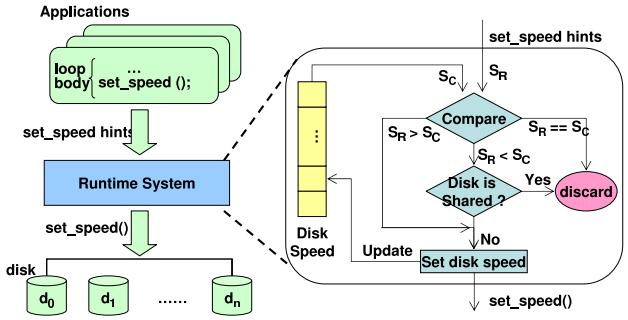


Fig. 6. Our proposed runtime system that handles compiler-inserted disk power management calls.

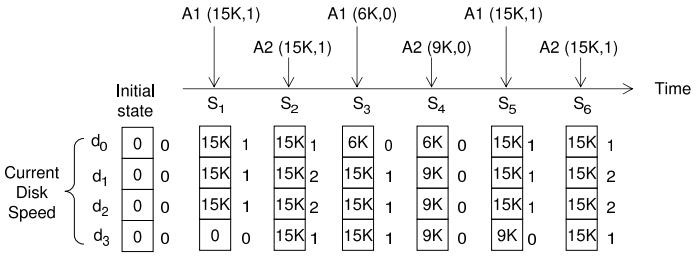


Fig. 7. An example that illustrates the selection of preferable disk speeds for a scenario that involves two concurrently-running applications. The disk layouts of the file accessed by A1 and A2 are (0, 3, 64) and (1, 3, 64), respectively. Each entry in the current disk speed array corresponds to the speed of each disk. The number next to each disk speed is the counter value associated with each disk, which indicates how many applications are currently using that disk.

by the compiler, and it should maintain the current speed and the requested one for every disk in the system. For this purpose, our runtime system maintains an array of disk speeds, each of which corresponds to the current speed of each disk. For example, assuming we have four disks in the system, {15K,15K,12K,9K} means that the current speed of d_0 and d_1 is 15K, and the current speeds of d_2 and d_3 are 12K and 9K, respectively. The decision for selecting the most appropriate speed for each disk depends primarily on the current disk speed and the requested disk speed, denoted as S_C and S_R respectively in Figure 6. If the current speed of a disk used by an application is S_C and another application issues a set_speed() call requiring speed S_R for the same disk, this request will be granted if and only if $S_C < S_R$. In other words, an application is allowed to speed up a disk but it is not allowed to slow down a disk if that disk is also being used at the same time by another application. To keep track whether a disk is shared or not, we attach a counter to each element of the disk speed array. The counter attached to a disk is increased if and only if a set_speed() call issued by an application and the F parameter to it is set to 1. Recall that, the F parameter in the set_speed() call is set to 1 when the compiler predicts that the disk should be spinning at the specified speed in order not to incur performance degradation. The counter is decreased when an application releases its use of disks by issuing the set_speed() call with the F parameter set to 0, or when the application terminates. If the disk is used by only one application (i.e., the counter attached to it is 1), a request to slow down the disk is granted by our runtime system. On the other hand, if the requested speed is equal to the current one, such a request is discarded.

Figure 7 shows an example of how our runtime system selects preferable disk speeds based on the hints issued by two applications, A1 and A2, running in parallel. The disk layouts of the files accessed by A1 and A2 are assumed to be (0, 3, 64) and (1, 3, 64), respectively, and it is assumed that we have four disks in the system. Initially, all the entries in the array of current disk speeds are set to zero, as depicted in Figure 7. The counter associated with each entry is also initialized to zero. In this example, each application sends three speed-setting calls (denoted as S_i in the figure) to the runtime system during execution. When our runtime system receives S_1, it updates the array of current disk speeds accordingly. Since the disk layout of the file accessed by A1 is (0, 3, 64), it sets the speed of the first three disks (i.e., d_0, d_1, and d_2) to 15K. When the runtime system receives S_2 from A2, it tries to set the speed of the disks accessed by A2, which are d_1, d_2, and d_3. The runtime system sets the speed of d_3 only, and the requests for the disks d_1 and d_2 are discarded because the requested speed of S_2 is the same as the current speed, which is 15K. The third and fourth requests need to be processed carefully because they demand slower speeds than the current one. In processing S_3, the runtime system first checks whether the disks accessed by A1 are shared or not. Since at this point only d_1 and d_2 are shared with A2 (i.e., the counters associated with d_1 and d_2 are not 1), the runtime system sets the speed of d_0 to the requested one, which is 6K (since this disk is not currently shared). After processing S_3, the corresponding counter is decreased accordingly. Similarly, when processing S_4, the runtime system allows slowing down the disks accessed by A2 to 9K because both A1 and A2 inform that the disks accessed by them are not used for a certain period of time. The remaining two requests, S_5 and S_6, on the other hand, are granted immediately because they require a higher disk speed than the current one.

B. Implementation

We implemented a prototype of our runtime system within PVFS2 [12], an open source parallel file system designed to run on Linux clusters. We modified three major parts of PVFS2 and MPICH2 [28] to support our runtime system approach. First, since the application code transformed by the compiler contains an explicit power management call, set_speed(), and this call is later used by our runtime system layer, we added this interface to ROMIO, which is the part of MPICH2 that contains the implementation of MPI-IO call interfaces to PVFS2. Since any number of executables compiled using the modified MPICH2 can make a set_speed() request to the runtime system simultaneously, we have to ensure that processing such requests is performed in an atomic fashion. To achieve this, we used a mutex-lock mechanism through the internal lock functions supported by PVFS2.

Second, the array of the current disk speeds is implemented as an internal data structure of the PVFS2 server. Since every PVFS2 server running on each node should know the current status of the disk speeds to make their own decisions, we added this data structure to the metadata server so that all PVFS2 I/O nodes can lookup the shared data structure when they make

the decision. Specifically, when a set_speed() call is received, the PVFS2 server checks to see if the specified disk speed can be set based on the decision made by the runtime module described in Figure 6. If the requested speed is granted by the runtime module, it is queued to the job threads of the PVFS2 server.

The last part of our modifications to PVFS2 is to emulate multi-speed disks. Since we do not have access to a multi-speed disk system, we need to emulate its behavior to evaluate our approach. There are two unique disk activities of multi-speed disks we have to emulate to obtain correct timing regarding disk accesses. The first activity is changing the speed of the disks, as triggered by our runtime system. The second one is handling disk I/O when the speed is set to a lower one than the maximum speed. To mimic the behavior of these activities, we used `nanosleep()`, a high resolution sleep function available under Linux/Unix. Specifically, whenever the runtime system makes a decision to change the speed of a particular PVFS2 I/O node (disk), we made that node sleep until the time specified by the argument to the nanosleep call. As explained in [7], the amount of time required for changing the speed of a disk depends on the difference between the current speed and the requested speed, and in our implementation we applied the technique in [7] to calculate this cost, and included it in all our results presented below. Similarly, the slowed I/O time due to the lower disk speed, which is set by the runtime system, is also imitated by intercepting every I/O thread[1] and invoking the required sleep calls before the thread terminates.

VI. Experimental Evaluation

A. Setup

We used the trace library component of Pablo [29], an I/O performance analysis toolkit, to collect the disk I/O traces for each application. We ran our applications on a Linux cluster of 16 dual processor nodes, AMD Athlon MP2000+ systems, connected through Ethernet and Myrinet. Each node of this system is configured as a PVFS2 I/O node and we installed MPICH2 in our execution environment. All applications that will be explained later in this section are MPI-IO based and use PVFS2 as the file system. We compiled them using the same compilers, Intel C and Fortran compiler 9.0, which MPICH2 was built with and ran them under the MPICH2 and PVFS2 environment.

To evaluate our approach and the prior ones to disk power management, we built a custom energy simulator. The disk energy model we used in our simulator is based on the data from the data sheets of the IBM Ultrastar 36Z15 disk [30]. Table I gives important simulation parameters. The I/O traces collected from Pablo [29] capture the duration of each I/O activity as well as the impact of slowing down or speeding up a disk. The cost of changing the speed of disk, Δ_t, to the different amount of RPM change, Δ_n, is also given in Table I. We calculated the energy consumption of each application based on this timing information fed to our energy simulator.

[1]In PVFS2, each I/O request is processed by the Trove thread spawned by the server process.

TABLE I Default System parameters.

Parameter	Default Value
Number of disks	8
Number of processors	4
Disk drive model	IBM36Z15
Storage capacity (GB)	36.7
Maximum disk speed (RPM)	15000
Striping unit (Kbytes)	64
Disk layout	(0, 8, 64)
Available disk speeds (RPM)	(15000, 12000, 9000, 6000)
Active power consumption (Watt)	(13.5, 10.73, 8.57, 7.03)
Idle power consumption (Watt)	(10.2, 7.43, 5.27, 3.73)
Δ_t (sec) when Δ_n (RPM) is (9K, 6K, 3K)	(6.54, 4.36, 2.18)

Energy consumption during each of different RPM transitions is based on the quadratic curve fitting in [7]. By default, all arrays are striped over all 8 disks.

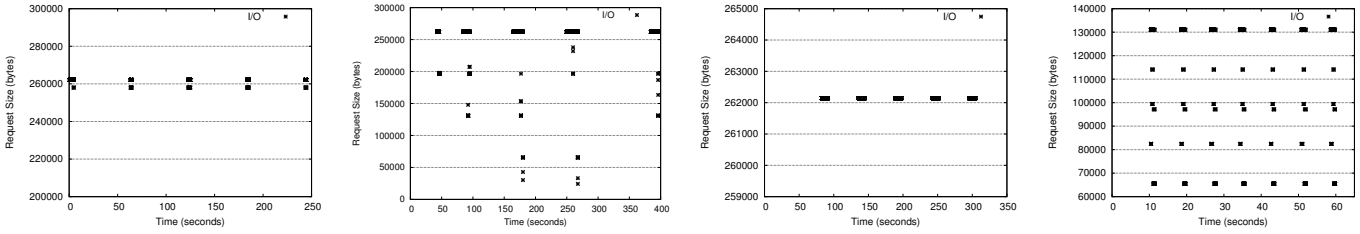
Table II gives a brief explanation of the applications used in this study. As can be seen from this table, the applications are taken from various sources and solve different types of problems from the scientific computing domain. The last two columns give the total amount of data manipulated by each application and its base execution time, using the default values of the system parameters listed in Table I (executing each application using four processors assuming the highest speed for all disks).

To evaluate the effectiveness of our approach, we made experiments with the following four schemes:

- **Base:** This version does *not* employ any disk power management strategy. All the disk energy and execution time numbers reported later in this section are given as values normalized with respect to this version.
- **COM:** This is the pure *compiler-based* approach proposed in [11]. In this scheme, the compiler estimates the disk idle and active periods by analyzing the application code and using the disk layouts exposed to it, and then inserts explicit disk speed-setting calls into the application. Note that the calls issued by this scheme go directly to the disk system, as opposed to our hints which go through the runtime system.
- **HW:** This is the pure *hardware-based* dynamic disk speed management scheme proposed in studies [1] and [7]. Depending on the observed variation on I/O response time, this scheme sets the speed of each disk to an appropriate level to save power. Note that a hardware scheme that employs spin-down disks [5], [6], [35] could also be used. However, we do not consider that option in this work, because the disk idle periods exhibited by these (and many other) scientific applications are very short, and consequently there are little chances of spinning the disks down and up (see Figure 8).
- **RT:** This is our runtime based approach proposed in this paper. The compiler extracts the disk access patterns of individual application considering disk layouts and generates speed-setting *hints*, and then the runtime system uses these hints and current status of the shared disks to make decisions regarding appropriate disk speeds for all applications.

TABLE II I/O APPLICATIONS USED IN OUR EXPERIMENT.

Name	Source	Language/Lib	Description	Data Size (MB)	Cycles (sec)
MxM	IBM [31]	C/MPI-IO	Originally MPI-IO/GPFS test program for out-of-core block matrix multiplication. It was modified to use PVFS2 through MPI-IO, and all matrices are assumed to be square.	191.8	244.8
S3aSim	Northwestern University [32]	C/MPI-IO	This is a parallel sequence search algorithm for evaluating various I/O strategies using MPI-IO. We compiled it to use collective I/O operations.	798.3	396.2
NIAL	MSE PSU [33]	F90/FFTW	This application quantitatively simulates phase transformation and coarsening in Ni-Al binary alloy in 3D using the KKS model using parallel programming.	720	308.9
BTIO	NPB2.4 [34]	F77/MPI-IO	This program implements the Block-Tridiagonal (BT) NPB problem, and employs MPI-IO. We compiled it with collective buffering.	419.43	483.9



(a) MxM (b) S3aSim (c) NIAL (d) BTIO

Fig. 9. I/O request patterns of each application throughout the whole execution time. Note that, in case of BTIO, we show only the first 65 seconds of total execution time to highlight the I/O access pattern; the remaining part until the completion of BTIO shows similar I/O behavior.

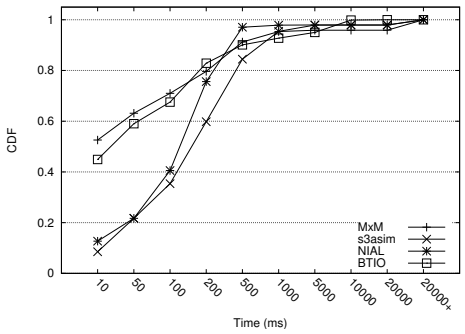
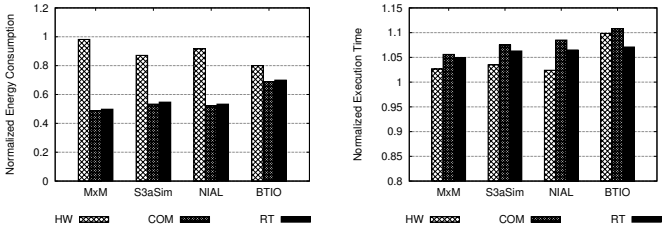


Fig. 8. CDF (Cumulative Distribution Function) for disk idle periods in our applications. An (x, y) point on a curve indicates that y% of the idle periods have a duration of x milliseconds (ms) or lower. The minimum amount of idle time required to compensate the cost of spinning down the disk and up is called the *break-even* time. Based on the numbers from the IBM36Z15 [30] disk data, the break-even time is 15.19 seconds. As most of idle periods are less then 1 second, this confirms that the spin-down disk is not a viable option for these applications. Instead, multi-speed disks are more promising.



(a) Normalized energy consumption. (b) Normalized execution time.

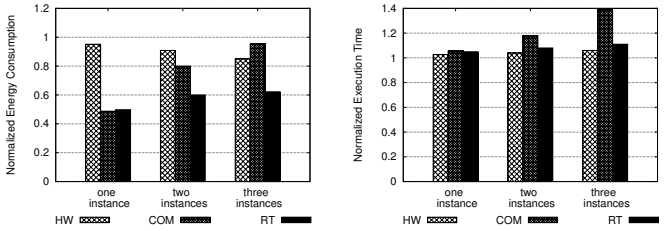
Fig. 10. Single application execution scenario.

B. Results

Before discussing the behavior of the different schemes for reducing disk energy consumption, let us first present the I/O patterns exhibited by the applications used in this study. The graphs in Figure 9 present the I/O patterns (obtained using Pablo [29]) of each application in isolation. In each plot, the x-axis shows timestamps and the y-axis indicates the size of I/O requests. As we can see from these plots, each application has explicit I/O phases which can take different amount of time and use various request sizes.

We start by presenting the results for the single application execution case. Figure 10 gives the normalized energy consumption and execution time results (with respect to the Base scheme) when each application is executed alone. One can make several observations from the results. First, the HW scheme brings an average energy saving of 14.5% across all applications with only 4.6% performance penalty. The second observation one can make is that the COM scheme achieves a significant energy savings, 44.2% on average over the Base scheme. This significant energy saving can be attributed to the proactive (speed-setting) decisions made by the compiler. Our next observation is that the energy savings brought by the RT scheme are very similar to those obtained using the COM scheme. More specifically, the RT scheme consumes only 1–2% more energy than the COM scheme. This is mainly due to the latency overhead incurred by our runtime system in processing set_speed() calls during execution. This result indicates that our runtime system performs very well in the single application execution scenario, with only minimal performance overhead.

In the next set of experiments we increased the number of applications running simultaneously to see how our approach behaves under multiple application scenario. Figure 11 shows the normalized energy consumption and execution time results when we increase the instances of MxM up to three. The results are normalized with respect to the Base case (i.e.,

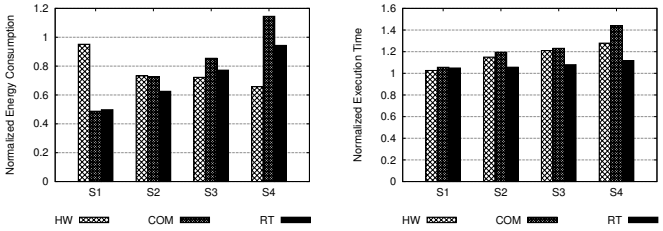


(a) Normalized energy consumption. (b) Normalized execution time.

Fig. 11. Multiple application execution scenario by increasing the instances of MxM up to three. The completion times for running two and three instances of MxM are 279.9 and 290.3 seconds, respectively. Due to caching effects, there is little increase in execution time when we increase the instances of MxM from two to three.

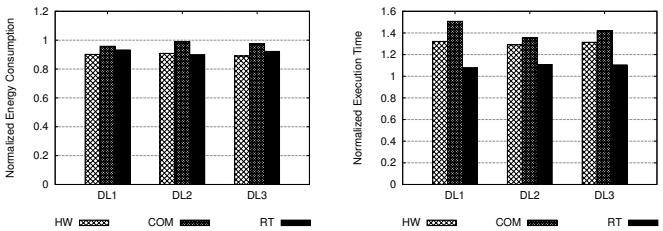
TABLE III MULTIPLE APPLICATION EXECUTION SCENARIO.

Scenario	Running Applications	Completion Time (sec)
S1	MxM	244.8
S2	MxM + S3aSim	464.0
S3	MxM + S3aSim + NIAL	371.9
S4	MxM + S3aSim + NIAL + BTIO	651.4



(a) Normalized energy consumption. (b) Normalized execution time.

Fig. 12. Multiple application execution scenario with different set of applications.



(a) Normalized energy consumption. (b) Normalized execution time.

Fig. 13. Multiple application execution scenario with different disk layouts (DLs). We used all four applications, each of which is configured to get striped across eight disks among fifteen available disks.

no power management); but, in this case, the execution time represents the completion time of the last (slowest) instance (of MxM), and the energy consumption includes all the disk energy consumed by all instances (i.e., the total energy consumption in the disk system). As we can see from Figure 11, when we increase the instances of MxM running concurrently, the energy savings brought by the HW scheme also increases slightly whereas the savings obtained through the COM and RT schemes decrease. The reason why the HW scheme saves more energy with more instances is that the periodic speed modulation by the disk drive itself comes to play with the mixed (interleaved) patterns of I/O requests. We can also observe that the HW scheme does not incur much performance slowdown, only 4.2% on average. However, the COM scheme performs poorly as we increase the number of instances of MxM, in terms of both energy and performance. Based on Figure 11(a), when three instances of MxM are executed at the same time, the COM scheme is slightly worse than the HW scheme in energy consumption. This is because the compiler-directed disk speed setting calls issued by one instance of MxM can be an untimely decision (e.g., slowing down a disk when it needs to be run at a higher speed) for the other instances of MxM that exercise the same set of disks (recall, from Table I that, by default, all arrays are striped over all 8 disks). The last observation we make is that the RT scheme always achieves the highest energy savings, 30.9% on average, considering all execution scenarios. These results clearly show that our approach is very successful with the multiple application execution scenario.

We next present results for four execution scenarios described in Table III. The second column of this table gives the applications executed simultaneously in the corresponding scenario and the third column gives the completion time, the time it takes for all the applications in the scenario finish their execution (under the Base scheme). As we can see from this table, the completion times for these different scenarios increase significantly when we execute more applications in parallel. Figure 12 shows the effect of three schemes we experimented with the scenario in Table III on the energy and execution times. In this figure, the bars for a given scenario is normalized with respect to the energy and execution times of the Base scheme. We can observe similar trends from these results as in the cases where we execute the multiple instances of the same application (see Figure 11). Specifically, while the HW scheme achieves increased energy savings when the number of applications is increased, the COM scheme does not perform well in saving disk energy. These results are expected because, from the compiler perspective, increasing the number of applications running in parallel has an impact on the performance and, ultimately, on the disk energy consumption, regardless of whether the instances running in parallel belong to the same application or not. In this execution scenario, the RT scheme achieves 19.4% and 39.9% energy savings over the HW and the COM schemes, respectively. As far as the performance is concerned, the RT scheme degrades the base execution time by less than 7.6%. The HW and COM schemes, on the other hand, incur 16.7% and 23.1% performance slowdowns, respectively. Note that the performance of the HW scheme under this execution scenario is much worse than the execution under the multiple instances of MxM (see Figure 11(b)). This is because the heuristic employed in the HW scheme modulates the disk speed based on the number of request (rather than time), and this modulation heuristic incurs more performance penalty when the number of I/O requests are increased.

In the last set of experiments, we varied the default striping of each file to assess the impact of different disk layouts on performance and energy consumption. In these experiments, we used the S4 scenario (see Table III) only. Recall that, all the results presented so far are obtained using 8 disks and

all files are striped across all available disks. In this new set of experiments, we increased the total number of disks that can be used for file striping to 15, and varied the Base disk for each array to effect its disk layout (see the disk layout triplet discussed in Section IV). The other two parameters, Striping_Factor and Striping_Unit, are fixed at 8 and 64Kbytes, respectively, as in the initial setup. We generated a random number between 1 and 15 to select the Base disk from which the file striping starts. Figure 13 gives the normalized energy consumption and execution time results for the S4 scenario with three different disk layouts. We see that the overall energy savings brought by our runtime system based approach are similar across the different layouts. This indicates that our approach can adapt very well to the different disk layouts. This is mainly because the compiler can extract the disk access patterns accurately with different disk layouts and provide the most preferable decision considering the underlying disk layouts. Our runtime system takes these preferable speeds and makes a globally acceptable decision for each and every disk. To summarize, these results show that our approach works very well when disks are shared and when they are not shared.

VII. Conclusion and Future Work

The main contribution of this paper is a runtime system support for software-based disk power management in the context of MPI-IO based scientific applications. In the proposed approach, the compiler provides important information on future disk access patterns and preferable disk speeds for each MPI-IO call. Our proposed runtime system uses this information to make decisions (regarding disk speeds) that would be agreeable to all applications running concurrently. Our main goal is to save as much energy as possible without slowing down any application significantly. To quantify the benefits of our approach, we implemented it using the PVFS2 file system. The results obtained so far from our experiments with four I/O-based scientific applications indicate significant energy savings over the pure hardware and software based schemes when multiple applications use the same set of disks concurrently. Our future work includes investigating a framework that integrates our approach with disk layout optimization in a unified manner.

Acknowledgments

This work is supported in part by NSF grants #0444158 and #0406340.

References

[1] E. V. Carrera, E. Pinheiro, and R. Bianchini, "Conserving Disk Energy in Network Servers," in *ICS'03*, June 2003, pp. 86–97.

[2] EMC Enterprise Storage System, "Symmetrix 3000 and 5000 Enterprise Storage Systems Product Description Guide," http://www.emc.com/, 1999.

[3] Maximum Institution Inc., "Power, Heat, and Sledgehammer," http://www.max-t.com/downloads/whitepapers/SledgehammerPowerHeat20411.pdf, 2002.

[4] F. Moore, "More Power Needed," *Energy User News*, November 2002.

[5] F. Douglis, P. Krishnan, and B. Bershad, "Adaptive Disk Spin-down Policies for Mobile Computers," in *Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, 1995, pp. 121–137.

[6] F. Douglis, P. Krishnan, and B. Marsh, "Thwarting the Power-Hungry Disk," in *USENIX'94*, 1994, pp. 292–306.

[7] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke, "DRPM: Dynamic Speed Control for Power Management in Server Class Disks," in *ISCA'03*, June 2003, pp. 169–179.

[8] E. Pinheiro and R. Bianchini, "Energy Conservation Techniques for Disk Array-Based Servers," in *ICS'04*, June 2004, pp. 66–78.

[9] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini, "Application Transformations for Energy and Performance-Aware Devicement Management," in *PACT'02*, September 2002, pp. 121–130.

[10] S. W. Son, G. Chen, M. Kandemir, and A. Choudhary, "Exposing Disk Layout to Compiler for Reducing Energy Consumption of Parallel Disk Based Systems," in *PPoPP'05*, June 2005, pp. 174–185.

[11] S. W. Son, M. Kandemir, and A. Choudhary, "Software-Directed Disk Power Management for Scientific Applications," in *IPDPS'05*, April 2005, p. 4b.

[12] PVFS2 Development Team, "Parallel Virtual File System, Version 2," http://www.pvfs.org/pvfs2-guide.html, September 2003.

[13] C. Gniady, Y. C. Hu, and Y.-H. Lu, "Program Counter Based Techniques for Dynamic Power Management," in *HPCA'04*, 2004, pp. 24–35.

[14] K. Okada, N. Kojima, and K. Yamashita, "A Novel Drive Architecture of HDD: "Multimode Hard Discdrive"," in *Proceedings of the International Conference on Consumer Electronics*, June 2000, pp. 92–93.

[15] 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," in *HPCA'04*, 2004, pp. 118–129.

[16] A. E. Papathanasiou and M. L. Scott, "Energy Efficient Prefetching and Caching," in *USENIX'04*, 2004, pp. 255–268.

[17] D. Li and J. Wang, "EERAID: Energy Efficient Redundant and Inexpensive Disk Array," in *Proceedings of the 11th workshop on ACM SIGOPS European workshop: beyond the PC*, 2004, p. 29.

[18] C. Weddle, M. Oldham, J. Qian, A.-I. A. Wang, P. Reiher, and G. Kuenning, "PARAID: The Gear-Shifting Power-Aware RAID," Department of Computer Science, Florida State University, Tech. Rep. TR-060323, 2006.

[19] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes, "Hibernator: Helping Disk Arrays Sleep Through the Winter," in *SOSP'05*, October 2005.

[20] L. Cai and Y.-H. Lu, "Power Reduction of Multiple Disks using Dynamic Cache Resizing and Speed Control," in *ISLPED'06*, 2006, pp. 186–190.

[21] D. Colarelli and D. Grunwald, "Massive arrays of idle disks for storage archives," in *SC'02*, July 2002.

[22] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-Performance, Reliable Secondary Storage," *ACM Computing Survey*, vol. 26, no. 2, pp. 145–185, 1994.

[23] S. Garg, "TFLOPS PFS: Architecture and Design of a Highly Efficient Parallel File System," in *SC'98*, 1998, pp. 1–12.

[24] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *FAST'02*, January 2002, pp. 231–244.

[25] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, 1999.

[26] J. M. May, *Parallel I/O for High Performance Computing*. Morgan Kaufmann Publishers, 2001.

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

[28] W. Gropp, E. Lusk, D. Ashton, P. Balaji, D. Buntinas, R. Butler, A. Chan, J. Krishna, G. Mercier, R. Ross, R. Thakur, and B. Toonen, "MPICH2 User's Guide, Version 1.0.5," December 2006.

[29] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed, "Input/Output Characteristics of Scalable Parallel Applications," in *SC'95*, 1995, p. 59.

[30] IBM, "Ultrastar 36z15 hard disk drive," http://www.hitachigst.com/hdd/ultra/ul36z15.htm, 2001.

[31] Jean-Pierre Prost, "MPI-IO/GPFS Test Program for Out-Of-Core Block Matrix Multiplication," http://www.mhpcc.edu/training/workshop2/mpi_io/samples/testmat.c.

[32] A. Ching, W.-C. Feng, H. Lin, X. Ma, and A. Choudhary, "Exploring I/O Strategies for Parallel Sequence Database Search Tools with S3aSim," in *HPDC'06*, June 2006, pp. 229–240.

[33] J. Zhu, L. Chen, J. Shen, and V. Tikare, "Coarsening kinetics from a variable mobility cahn-hilliard equation – application of semi-implicit fourier spectral method," *Phys. Review E*, vol. 60, no. 4, pp. 3564–3572, 1999.

[34] P. Wong and R. F. V. der Wijngaart, "NAS Parallel Benchmarks I/O Version 2.4," NASA Advanced Supercomputing Division, Tech. Rep. NAS-03-002, January 2003. [Online]. Available: http://www.nas.nasa.gov/New/Techreports/2003/PDF/nas-03-002.pdf

[35] R. K. K. Li, P. Horton, and T. Anderson, "A Quantitative Analysis of Disk Drive Power Management in Portable Computers," in *USENIX'94*, 1994, pp. 279–292.