Software-Directed Disk Power Management for Scientific Applications *

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

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

Abstract

Disk power consumption is becoming an increasingly important issue in high-end servers that execute large-scale data-intensive applications. In particular, array-based scientific codes can spend a significant portion of their power budget on the disk subsystem. Observing this, the prior research proposed several strategies, such as spinning down to low-power modes or adjusting the speed of the disk in lower RPM, to reduce power consumption on the disk subsystem. A common characteristic of most of these techniques is that they are reactive, in the sense that they make their decisions based on the disk access patterns observed during execution. While such techniques are certainly useful and the published studies reveal that they can be very effective in some cases, one can conceivably achieve better results by adopting a proactive scheme.

Focusing on array-intensive scientific applications, this paper makes two important contributions. First, it presents a compiler-driven proactive approach to disk power management. In this approach, the compiler analyzes the application code and extracts the disk access pattern. It then uses this information to insert explicit disk power management calls in the appropriate places in the code. It also preactivates a disk (placed into the low-power mode) before it is actually needed to eliminate the potential performance impact of disk power management. The second contribution of this paper is a code transformation approach that can be used to increase the savings coming from a disk power management scheme (whether reactive or proactive). Our experimental results with several scientific application codes show that both the proactive disk power management approach and the disk layout aware code transformations are beneficial from both power consumption and execution time perspectives.

* This work is supported in part by NSF Grants #0444158, #0406340, and #0093082.

1. Introduction

Power consumption of large servers has recently been a popular research topic. There are at least three reasons for that. First, these servers are power hungry systems as documented by the prior work, which indicates that they can consume several mega-watts of power [4, 3]. Second, the cooling systems required for these servers can be extremely costly, which in turn contributes to expensive electrical bills [2]. Third, since high power consumption is also harmful to environment, there is a strong motivation from an environmental perspective as well to focus on power consumption.

The prior efforts that target at reducing power/energy consumption of servers can be broadly divided into three categories. The efforts in the first category [9] are CPU-centric studies and consider techniques such as shutting down unused CPUs and voltage scaling. The second category [3, 13, 5] considers different components of the servers such as communication links, CPUs, and memories. The third group, which our approach also belongs to, focuses on the disk subsystem of the servers and considers disk shutdown and adaptive speed-setting policies at the architecture and OS (operating system) levels. Note that, disk subsystem can be responsible for a large percentage of the total power budget of a system, as noted in several prior studies such as [10] and [16]. The representative studies for disk power management include spinning down to low-power modes [7, 8] (most traditional power management techniques, referred to as TPM in this paper, utilize this scheme) , dynamic RPM (DRPM) [10], and Popular Data Concentration (PDC) [16]. An important common characteristic of these previous disk power management techniques is that they are *reactive,* in the sense that they make their decisions based on the disk access patterns observed during execution. While such techniques are certainly useful and the published studies reveal that they can be very effective in some cases, one can conceivably achieve better results by adopting a *proactive* approach.

Focusing on scientific applications that manipu-

late large, disk-based datasets (mostly in the form of multi-dimensional arrays), this paper proposes such a proactive scheme based on analyzing the application code and extracting data/disk access pattern information. This information is in turn used during execution to proactively spin down/spin up disks (when used in conjunction with TPM-capable disks), or change their rotation speeds (when used in conjunction with DRPM-capable disks). In the proposed approach, we enlist the compiler's help to analyze and extract the data/disk access pattern, and decide the most suitable disk power management strategy. There are at least two advantages of such a proactive scheme over the reactive approaches proposed by the prior research. First, since for array-based codes this approach can identify disk idle periods accurately, this information can be used to select the most appropriate power management strategy/mode (e.g., the most suitable disk speed in DRPM), and this in turn helps reduce the energy consumption on the disk subsystem. Second, the compiler-directed scheme can also determine when an idle disk will be requested again and *pre-activate* the disk (by spinning it up) before it is actually needed; this helps reduce performance penalties (e.g., due to spinning up or due to changing the speed). A potential drawback of the proactive scheme is that we need the source code to analyze and extract data access pattern. Therefore, in our work, we restrict ourselves to array-based scientific applications where source codes are accessible to and analyzable by an optimizing compiler. The proposed proactive scheme can be used in conjunction with both TPM and DRPM.

The second contribution of this paper is a *compiler-directed code transformation approach* to improve the effectiveness of the proactive power management strategy. In this approach, the code is restructured (automatically) based on the layout of the data on the disk subsystem to increase the disk inter-access times, which is beneficial for reactive and proactive schemes alike. Lastly, we present a comprehensive experimental analysis of the proposed proactive scheme as well as the code transformation approach using a simulation platform, and demonstrate their effectiveness through a set of scientific codes that manipulate disk-resident datasets. We also discuss how different disk layouts of data can affect the benefits obtained from the proactive scheme. It should be noted that the proactive power management schemes have been employed by the prior research in the context of CPUs [9], main memories [6, 15], and network components [5]. Therefore, our work can also be considered as an adaptation of such generic schemes to the disk domain, whose power consumption is becoming an increasing concern in high-performance computing for both hardware designers and software writers.

The rest of this paper is organized as follows. Section 2 gives a brief overview of TPM and DRPM, the two previously-proposed approaches to disk power management. Section 3 presents the details of our compiler-driven approach, focusing in particular on disk access pattern extraction and code modification to insert explicit disk power management calls. Section 4 explains our experimental setup, benchmarks, and the different versions tested. Section 5 reports experimental data that demonstrate the effectiveness of the compiler-directed scheme. Section 6 discusses and quantifies the impact of code transformations (restructurings). Finally, Section 7 concludes the paper with a summary of our major contributions.

2. TPM and DRPM

Disk power management has been extensively studied in the context of laptop/desktop disks [7, 8]. Many current disks have several operating modes such as active, idle, and one or more low-power operating modes. In traditional disk power management techniques (denoted as TPM in this paper), if the detected disk idle period is longer than a certain amount of time, called the *idleness threshold*, the disk is spun down to the low-power mode. The disk remains in the low-power mode until it receives the next request. Note that this strategy typically incurs performance slowdown because the disk should first spin up to service the upcoming request. The time it takes to spin up/down a disk is called the *spin-up/down time*. Therefore, in TPM, choosing the idleness threshold, by making use of either fixed or adaptive threshold based strategies, is crucial in managing both disk energy and performance. While TPM is an effective approach in the domain of laptop/desktop systems, recent studies [10, 2] demonstrated that it is not an appropriate choice for large servers and cluster based systems.

Since the disk spin-up/down time is much greater in the server class disks (as compared to laptop/desktop systems) and exploiting idle time is infeasible, Gurumurthi et al proposed dynamic RPM (DRPM) [10]. The DRPM technique is similar, in principle, to CPU voltage scaling technique in that it dynamically changes the RPM step (the rotation speed of the disk) and can service a request with a reduced speed, provided that the disk hardware/controller supports several RPM steps, based on the I/O workload. It has been observed that DRPM can save a significant amount of disk power in the presence of server workloads where, in general, exploiting idle time is not viable option if we restrict ourselves to only TPM. In addition, since the RPM modulation time from one level to another is usually much smaller than typical spin-up/down times, the resulting performance degradation is also small compared to the TPM-based schemes. A similar technique to DRPM has been proposed and evaluated in [2]. When there is no confusion, in the rest of the paper, we use the term "low-power mode"

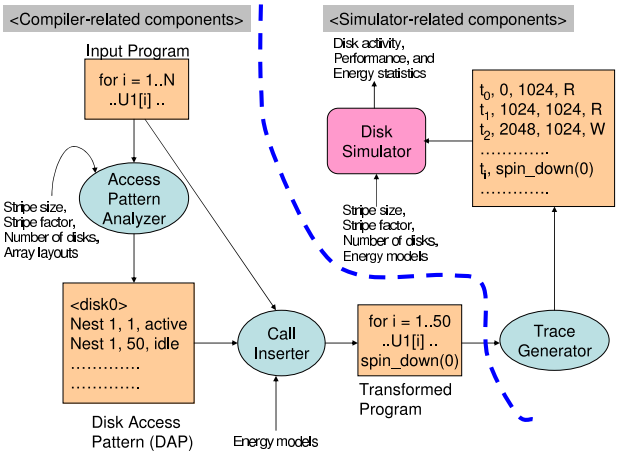

Figure 1. High-level view of our approach.

to denote either a disk which is spun down (in TPM) or a disk whose speed is set to a lower value than the maximum speed supported (in DRPM).

The effectiveness of these prior efforts on disk power management can be increased by analyzing the application code and exploiting the disk idle/active periods determined by the compiler. In addition, applying code and disk layout optimizations can lead to further improvements. In the rest of this paper, we study these issues in detail.

3. Our Approach

Our overall approach is depicted in Figure 1. In this section, we explain the compiler-related part that contains three important components: the compiler-analysis to identify disk accesses, the *disk access pattern* (DAP), and the insertion of explicit power management calls in the code.

In order to determine the disk access pattern, we need two types of information: data access pattern and disk layout of array data. The data access pattern indicates the order in which the different array elements are accessed, and is extracted by the compiler by analyzing the source code of the application. Since a number of compiler optimizations, e.g., those target data locality and parallelism, already make use of the results of this type of analysis (e.g., see [19] and the references therein), we do not present its details here. To determine which particular disks are being accessed, the compiler also needs the layout of array data (i.e., the file that holds the array elements) on the disk subsystem. In this context, the disk layout of an array (which is stored in a file) is specified using a 3-tuple:

(starting_disk, stripe_factor, stripe_size).

The first element in this 3-tuple indicates the disk from which the array is started to get striped. The second element gives the number of disks used to stripe the data, and the third element gives the stripe (unit) size. As an example, in Figure 2(b), array `U1` is striped over all four disks

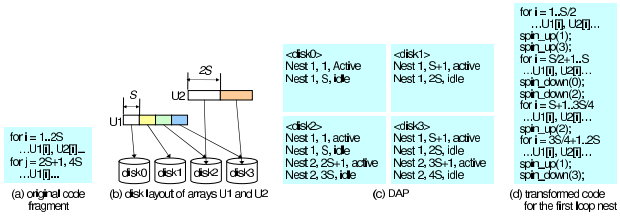

Figure 2. An example application of our approach.

in the figure. Assuming that the stripe size is S and the total array size is $4S$ (for illustrative purposes), the disk layout of this array can be expressed as $(0, 4, S)$. To illustrate the process of identifying the disk accesses, let us consider the code fragment in Figure 2(a). During the execution of the first loop nest, this code fragment accesses the array elements `U1[1]`, `U1[2]`, ..., `U1[2S]` and `U2[1]`, `U2[2]`, ..., `U2[2S]`. Consequently, for array `U1`, we access the first two disks (disk0 and disk1); and for array `U2`, we access only the third disk (disk2). Note that, the several current file systems and I/O libraries for high-performance computing support calls available to convey them the disk layout information when the file is created. For example, in PVFS [17], we can change the default striping parameter 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 access pattern it extracts to determine the disk access pattern. On the other hand, if the file is already created on the disk subsystem, the layout information can be passed to the compiler as a command line parameter.

The DAP lists, for each disk, the idle and active times in a compact form. An entry for a given disk looks like:

$$\begin{array}{lll} < \text{Nest 1}, & \text{iteration 1}, & \text{idle} > \\ < \text{Nest 2}, & \text{iteration 50}, & \text{active} > \\ < \text{Nest 2}, & \text{iteration 100}, & \text{idle} > \end{array}$$

We see from this example DAP that, the disk in question remains in the idle state (not accessed) until the 50th iteration of the second nest. It is active (used) between the 50th iteration and the 100th iteration of the second nest, following which it becomes idle again, and remains so for the rest of execution. For the example code fragment in Figure 2(a) and the disk layouts illustrated in Figure 2(b), Figure 2(c) gives the DAPs for each of the four disks in the system. The last component of our compiler-driven strategy is responsible from inserting explicit disk power management calls in the code. It is important to note that a DAP is

given in terms of loop iterations. In order to determine the appropriate places in the code to insert explicit power management calls, we need to interpret the loop iterations in terms of cycles, which can be achieved as follows. The cycle estimates for the loop iterations are obtained from the actual measurement of the program execution by using a high-quality timer called *gethrtime*, which is available on the UltraSPARC-based systems. Using the measured execution time (in nanoseconds) and given the machine's clock rate, we estimate the number of cycles per each loop iteration.

Once we determine the estimated disk idleness (in terms of cycles), if this idleness is larger than the *break-even threshold*, i.e., the minimum amount of idle time required to compensate the cost of either spinning down in a TPM disk or changing RPM speeds in a RPM disk, the compiler inserts an appropriate power management call in the code depending on the underlying method used (e.g., TPM versus DRPM). The format of this call is as follows:

$$\begin{cases} \texttt{spin\_down(disk}_i\texttt{)} & : \text{ TPM disks} \\ \texttt{set\_RPM(rpm\_level}_j\texttt{,disk}_i\texttt{)} & : \text{ DRPM disks,} \end{cases}$$

where $\texttt{disk}_i$ is the disk id, and $\texttt{rpm\_level}_j$ is the jth RPM level (i.e., disk speed) available. Since a DAP indicates not only idle times but also active times anticipated in the future, we can use this information to preactivate disks that have been either spun down by a `spin_down` call or set to lowest RPM level by a `set_RPM` call. To determine the appropriate point in the code to spin up the disk, we take into account the spin-up time (delay) of the disk. Specifically, the number of loop iterations before which we need to insert the spin-up (pre-activation) call can be calculated as:

$$d = \lceil \frac{T_{su}}{s + T_m} \rceil \quad (1)$$

where *d* is the pre-activation distance (in terms of loop iterations), T_{su} is the expected spin-up time, T_m is the overhead of a `spin_up` call or a `set_RPM` call, and *s* is the number of cycles in the shortest path through the loop body. Note that, T_{su} is typically much larger than *s*. We also stripe-mine the loop, because it is unreasonable to unroll the loop to make explicit the point at which the spin-up call is to be inserted. The format of the call that is used to pre-activate (spin up) a disk is as follows:

$$\texttt{spin\_up(disk}_i\texttt{)},$$

where as before $\texttt{disk}_i$ is the disk id. Note that, in DRPM, we can use same `set_RPM` call with maximum RPM speed as a parameter. For our running example, Figure 2(d) shows the compiler-modified code with the `spin_down` and `spin_up` calls. Note that, if we do not use pre-activation, the disk is automatically spun up when an access (request) comes; but, in this case, we incur the associated spin-up delay fully. The purpose of the disk pre-activation is to eliminate this performance penalty.

Table 1. Default simulation parameters.

Parameter	Value
Parameters common to TPM and DRPM	
Disk Model	IBM Ultrastar 36Z15
Interface	SCSI
Storage Capacity	18 GB
RPM	15,000
Average seek time	3.4 msec
Average rotation time	2 msec
Internal transfer rate	55 MB/sec
Power (active)	13.5 W
Power (idle)	10.2 W
Power (standby)	2.5 W
Energy (spin down: idle → standby)	13 J
Time (spin down: idle → standby)	1.5 sec
Energy (spin up: standby → active)	135 J
Time (spin up: standby → active)	10.9 sec
Parameters specific to DRPM	
Maximum RPM level	15,000 RPM
Minimum RPM level	3,000 RPM
RPM Step-Size	1,200 RPM
Window size	30
Striping Information	
Stripe unit (stripe size)	64 KB
Stripe factor (number of disks)	8
Starting iodevice (starting disk)	0

To evaluate our proposed compiler-directed proactive approach to disk power management, we wrote a trace generator and a disk power simulator (see Figure 1). The details of the trace generator and the power simulator are discussed in the next section.

4. Experimental Platform

4.1. Setup and Benchmarks

To generate disk access patterns for each benchmark program, we implemented a trace generator. The cycle estimates for the loop nests were obtained from 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 striping information (see the corresponding part in Table 1). Based on these disk parameters, the simulator determines which I/O nodes it should access when it reads an I/O request. 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. In our simulator, the striping information is provided in an external file along with other parameters. The default simulation parameters are given in Table 1.

In order to evaluate our approach and the prior proposals to disk power management, we developed a trace-driven simulator, which is similar to DiskSim [1]. The simulator is driven by externally-provided disk I/O request traces, which are generated, as explained earlier, from the compiler-transformed codes. Each I/O request is composed of the four parameters: request arrival time (in milliseconds), start block number, request size (in bytes), and request type (read or write).

Table 2. Benchmarks and their characteristics.

Benchmark Name	Data Size (MB)	Num of Disk Reqs	Base Energy (J)	Execution Time (ms)
168.wupwise	176.7	24,718	20835.96	248790.00
171.swim	96.0	3,159	2686.79	32088.98
172.mgrid	24.7	12,288	10600.54	126651.12
173.applu	54.7	7,004	5875.11	70142.24
177.mesa	24.0	3,072	2667.00	31869.54
178.galgel	16.0	2,048	1715.37	20478.80

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 datasheet of the IBM Ultrastar 36Z15 [12], and are given Table 1. Because we are primarily interested in the performance and energy consumption of the disk subsystem, we assume that other performance enhancement techniques like I/O prefetching are not employed.

For DRPM, we obtained statistics using the model described in [10]. The simulation parameters specific to DRPM are also given in Table 1. Besides the parameters and values given in Table 1, we obtained the RPM transition time and energy consumption at each RPM level as well. For the energy consumption at each RPM transition, we conservatively assume that the energy consumed during transition is the same as that of the faster RPM level involved in the transition. In DRPM, each disk can transition from one RPM level to another based on the response time change in the *n*-request windows as suggested in [10]. We used the same heuristic algorithm in implementing DRPM for both upper and lower tolerance. However, we used a smaller window size (30) since our evaluation considers one benchmark program at a time, and the resulting number of I/O requests is comparatively small. 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.

Table 2 gives the set of array-based benchmark codes used in this study. These benchmarks were selected randomly from the Specfp2000 benchmark suite [18]. We made the data manipulated by these benchmarks disk resident. As a result, each array reference causes a disk access unless the data is captured in the buffer cache. Also, to complete our simulations within a reasonable amount of time, we focused only on time-consuming loop nests from these applications. Specifically, from each application, we selected the nests whose cumulative I/O time account for at least 90% of the total I/O time of the application. The second column in Table 2 gives the total dataset size manipulated by the selected 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* 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.

4.2. Disk Power Management Schemes

To compare different approaches to disk power management, we implemented and performed experiments with different schemes:

- 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 values normalized 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 [7] and [8], described in Section 2.
- Ideal TPM (ITPM): This is the ideal version of the TPM strategy. In this scheme, we assume the existence of an *oracle predictor* for detecting idle periods. Consequently, the spin-up/down activities are performed in an optimal manner; i.e., the disk is not spun down unless the idleness duration is large enough so that one can save power. While one can expect better performance/energy behavior with this scheme as compared to the TPM, it has still the same drawback of not being able to useful when the idle periods are small.
- DRPM: This is the dynamic RPM strategy proposed in [10], which is described earlier in Section 2.
- Ideal DRPM (IDRPM): This is the ideal version of the DRPM strategy. In this scheme, we assume the existence of an *oracle predictor* for detecting idle periods, as in the ITPM case. Consequently, the disk speed to be used is determined optimally. This does not just maximize energy savings on the disk subsystem, but also eliminates the potential performance penalties.
- Compiler-Managed TPM (CMTPM): This corresponds to our compiler-driven approach when it is used with TPM. The compiler estimates idle periods by analyzing code and considering disk layouts, and then makes spin-down/up decisions based on this information.
- Compiler-Managed DRPM (CMDRPM): This corresponds to our compiler-driven approach when it is used with DRPM. The compiler estimates idle periods by analyzing code and considering disk layouts, and then selects the best disk speed to be used based on this information.

It must be emphasized that, the ITPM and IDRPM schemes are *not* implementable. The reason that we make experiments with them is that we want to see how close our

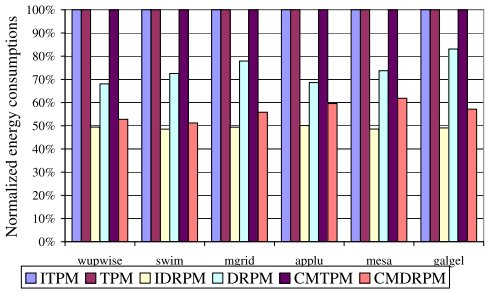

Figure 3. Normalized energy consumptions.

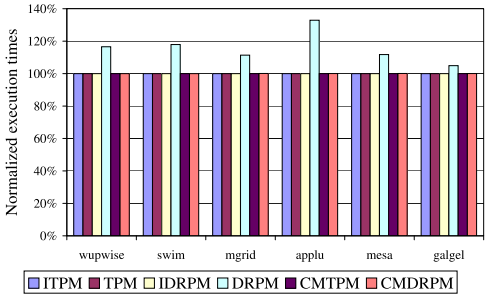

Figure 4. Normalized execution times.

compiler-based schemes come close to the optimal. All necessary code modifications are automated using the SUIF infrastructure [11].

5. Empirical Analysis

5.1. Base Results

The graph in Figure 3 gives the energy consumption of our benchmarks under the different schemes described earlier. One can make several observations from these results. First, as the idle times exhibited by the benchmark used are much smaller in length, the TPM version (ideal or otherwise) does not achieve any energy savings. Second, while the DRPM version generates savings (26% on average), the difference between it and the IDRPM is very large; the latter reduces the energy consumption by 51% when averaged over all benchmarks in our suite. This shows that a reactive strategy is unable to extract the potential benefits from the DRPM scheme. Our next observation is that the CMDRPM scheme brings significant benefits over the DRPM scheme, and improves the energy consumption of the base scheme by 46%. In other words, it achieves energy savings that are very close to those obtained by the IDRPM strategy. These results demonstrate the benefits of the compiler-directed proactive strategy.

It is to be noted, however, that the energy consumption is just one part of the big picture. In order to have a fair comparison between the different schemes that target disk power reduction, we need to consider their performances (i.e., execution times/cycles) as well. The bar-chart in Figure 4 gives the normalized execution times (with respect to the base version) for the different schemes evaluated. The reason why the TPM-based schemes do not incur any performance penalty is that they are not applicable, given the short disk idle times discussed earlier. When we look at the DRPM-based schemes, we see that the conventional DRPM incurs a performance penalty of 15.9%, when averaged over six benchmarks. We also see that the CMDRPM scheme incurs almost no performance penalty. The main reason for this is that this scheme starts to bring the disk to the desired RPM level before it is actually needed, and the disk becomes ready when the access takes place. This is achieved by accurate prediction of disk idle periods. These results along with those presented in Figure 3 indicate that the compiler-directed disk power management can be very useful in practice, in terms of both energy consumption and execution time penalty. Specifically, as compared to the reactive DRPM implementation, this scheme reduces the disk power consumption and eliminates the performance penalty. To better explain why the CMDRPM comes close to the IDRPM, we give in Table 3 the percentage of time that CMDRPM mispredicts the optimal disk speed, as compared to IDRPM. To collect this data, we recorded the RPM level used for each idleness for both IDRPM and CMDRPM. We see that the percentage mispredictions are not very large, which explains the success of the compiler-driven scheme.

Table 3. Percentage of mispredicted disk speeds.

	wupwise	swim	mgrid	applu	mesa	galgel
CMDRPM	6.78	5.14	13.02	18.97	27.35	15.9

5.2. Sensitivity Analysis

The magnitude of the benefits obtained by the proactive strategy depends on a number of parameters such as the stripe factor and the stripe size. In this subsection, we vary the values of these parameters to see how our savings are effected. For illustrative purposes, we choose one benchmark, *swim*, and conducted all sensitivity analysis on that. Figures 5 and 6 give the normalized energy consumptions and execution times, respectively, with different stripe sizes. The values of the all other simulation parameters are as given in Table 1. We see from these results that the energy savings brought by CMDRPM are consistent across wide range of stripe sizes. We also see that the compiler-based approach to disk power management does not increase the original execution times for the stripe sizes tested. In contrast, the behavior of the conventional DRPM becomes really worse when we increase in the stripe size. This can be explained as follows. As the stripe size increases and the access pattern remains sequential, this increases the service duration for a particular disk, i.e., more I/O requests go to the same disk. The controller then tries to bring the current RPM level to a lower level since the current workload is not heavy. This incurs a slowdown in response times for the

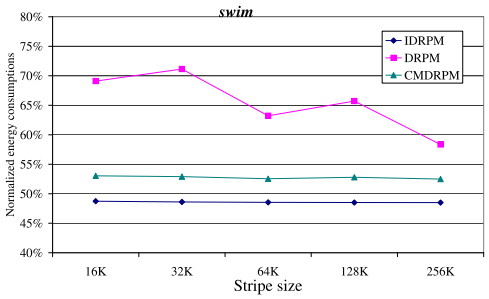

Figure 5. Energy consumption with different stripe sizes.

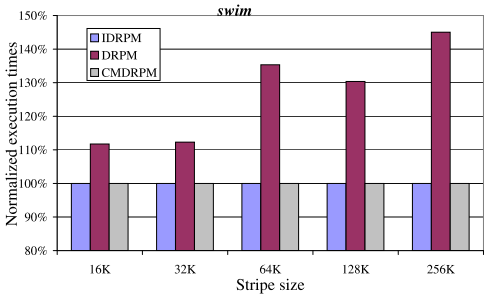

Figure 6. Execution times with different stripe sizes.

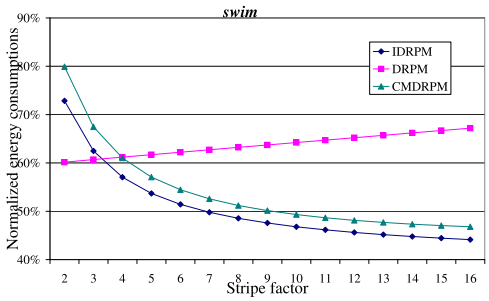

Figure 7. Energy consumption with different stripe factor.

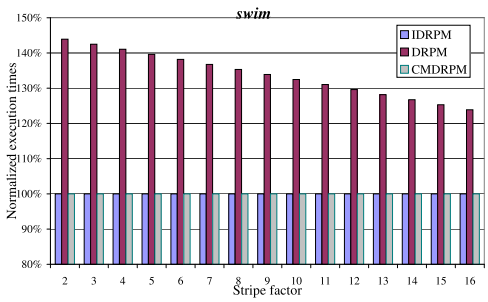

Figure 8. Execution times with different stripe factor.

next *n* requests before the controller restores the RPM level to a higher level to compensate for the previous slowdown in the response time. Since this trend holds as the stripe size increases, we can conserve energy consumptions, whereas the performance becomes worse with the larger stripe sizes.

The next parameter whose variation we study is the stripe factor (the number of disks). Figures 7 and 8 give the normalized energy and execution time results, respectively, with different stripe factors. As before, all other parameters are set to their default values given in Table 1. One can see from these results that the CMDRPM scheme generates more savings with the increased number of disks. This is because adding disks to the system increases the energy consumption of the base scheme dramatically. However, both the IDRPM and CMDRPM take advantage of the extra idle periods generated by these additional disks. We see that, from both energy consumption and performance angle, CMDRPM remains very close to the IDRPM.

6. Impact of Code Transformations

Our discussion and experimental evaluation so far made a case for compiler-directed proactive disk energy management. In this approach, the only modification made to the application code was the insertion of explicit power management calls to spin up and down disks, or to set disk speeds. However, one can potentially achieve better energy savings by restructuring application code, i.e., by modifying its data (and consequently disk) access pattern. While it is known from the optimizing compiler research [19] that loop transformations are very effective in optimizing data locality (mainly cache behavior) and iteration-level parallelism, there is no published study, to our knowledge, that studies the impact of such optimizations on disk energy consumption.

6.1. Transformation Approach

It is possible to increase the effectiveness of disk power management techniques by using loop distribution (fission). An important point to note here, though, is that the loop distribution in this case should be applied with care, taking into account the layout of data on the disk subsystem. That is, unlike the case with the conventional compilation frameworks that target data cache locality, in our context a loop transformation should be applied in a layout-sensitive manner. Considering underlying disk layouts, one can envision a compilation scheme that combines code restructuring and data layout optimizations under a unified setting. That is, the compiler can determine the most suitable disk layout of data along with the accompanying loop transformation. As has been discussed in Section 3, the disk layout of array data can be controlled by three parameters: stripe size, stripe factor, and starting disk (starting iodevice). The algorithm sketched in Figure 11 determines the necessary loop distribution (for each nest) and the corresponding

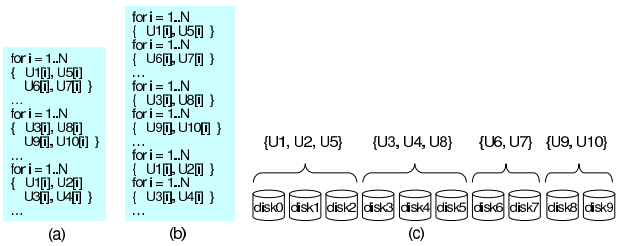

Figure 9. Loop fission example. (a) original code fragment. (b) transformed code fragment. (c) disk allocation for array groups.

disk layout (for each array) for a given program. In informal terms, it operates as follows. It visits each nest in the application code. For each nest, it considers a loop distribution such that the newly-generated loops after the distribution access disjoint sets of arrays as much as possible. It then forms *array groups*. The arrays in a group are the ones that are accessed by the same set of statements. After this step, each array group is assigned a disjoint set of disks. In our current implementation, we distribute the available disks across the array groups based on the total amount of data in each group; i.e., more data an array group has, more disks it is assigned in a proportional manner. Note that, what this algorithm is essentially trying to achieve is to place the disjointedly-accessed arrays into different array groups (and eventually to different disks), so that when one group is being accessed during execution the disks that hold the arrays of the other groups can be placed into the low-power mode. Figure 9 illustrates an example application of this algorithm. In the original code fragment shown in Figure 9(a), three loop nests access a total of 10 arrays (`U1`, `U2`, `U3`, . . . , `U10`). Assuming that all the arrays are of the same size and each loop nest has the same iteration count, our approach forms four array groups, {`U1`, `U2`, `U5`}, {`U3`, `U4`, `U8`}, {`U6`, `U7`}, and {`U9`, `U10`}. Note that, `U2` and `U5` belong to the same group, as they are coupled via array `U1`. These four array groups are assigned to the disks as shown in Figure 9(c). The fissioned code is given in Figure 9(b). Now, for example, when the application executes the first loop in the transformed code, the execution accesses only the first three disks; one can save a significant amount of energy by putting the unused disks in the low-power mode.

Another loop-based transformation, loop tiling [14], can also be used for increasing the effectiveness of disk power management. What this transformation essentially does is to restructure a loop nest (by dividing it into iteration tiles) in such a fashion that, at any given time, a single data block (from the array) is accessed, thereby exploiting the data reuse within the block. In tiling, the transformed loop nest has two types of loops: tile iterators and element iterators. The tile iterators iterate over a given iteration tile, whereas the element iterators operate over the members of a given

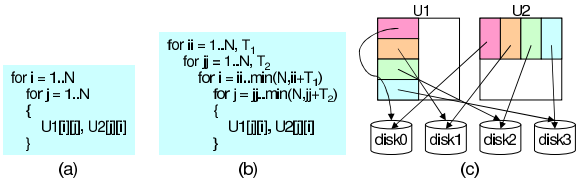

Figure 10. Loop tiling example. (a) original code fragment. (b) transformed code fragment. $T_1 \times T_2$ is the tile size. (c) tile-to-disk mapping.

tile. Therefore, if done appropriately, for a given execution of the tile iterators, the element iterators access only specific blocks of data. In our context, if we set the block size used in tiling to the disk stripe size, after tiling, at a given time the execution operates on certain set of blocks (typically one block from each array). As in the case of loop distribution, it is required to collocate the blocks that are operated at the same time, and thus this transformation also helps increase the energy savings of TPM and DRPM. A sketch of our layout-aware loop tiling algorithm for reducing disk energy consumption is given in Figure 12. An example application of this algorithm is illustrated in Figure 10. Figure 10(a) shows the original code fragment, and Figure 10(b) gives the tiled version. Note that, in this transformed code, `ii` and `jj` are the tile iterators, and `i` and `j` are the element iterators. Figure 10(c) shows the tile-to-disk assignment. It should be noted that, in order to achieve this assignment, array `U2` needs to be layout-transformed (from row-major to column-major). After this assignment, when we are working with a specific `ii`, `jj` pair (in Figure 10(b)) during execution, we access two specific tiles from arrays `U1` and `U2`, and the disks that do not hold these tiles can be placed into the low-power mode to save energy. Note that, unlike the loop distribution algorithm, the tiling algorithm explained here operates only for a single nest. Therefore, in our current implementation, we applied it only to the most costly nest (as far as disk energy consumption is concerned) from each application. As a result, the layout determined based on this most costly nest may not be preferable for the remaining nests. However, our experiments show that in several cases this algorithm generates disk energy savings. Extending this tiling approach to multiple nests is in our future agenda.

6.2. Results

To evaluate the impact of loop distribution and loop tiling on the effectiveness of TPM and DRPM, we performed a set of experiments with the following new versions:

- LF: The loop fission based version that does not use disk layout optimization.
- TL: The loop tiling based version that does not use disk layout optimization.

- LF+DL: The layout-aware loop fission version based on the algorithm given in Figure 11.
- TL+DL: The layout-aware tiling version based on the algorithm given in Figure 12.

It is to be noted that, each of these versions can be combined with TPM, ITPM, DRPM, IDRPM, CMTPM, or CMDRPM. That is, the proposed code (and data layout) optimization approach can be useful with both reactive and proactive schemes. The reason that we make experiments with (layout-oblivious) versions such as LF and TL is that we want to see whether conventional loop distribution and tiling (i.e., without considering disk layouts) can be any useful in reducing disk energy. Also, the DL denotes the different concepts in LF+DL and TL+DL. Specifically, in the LF+DL version, DL indicates the division of arrays across the disks, whereas in the TL+DL version, it indicates the layout transformation and tile-to-disk mapping.

The normalized energy consumptions with the different schemes are given in Figure 13. As before, all the results are normalized with respect to the base version. We can make the following observations based on these results. First, the LF and TL versions do not perform well. This is not surprising since they do not consider disk layout, and indiscriminately fissioning or tiling the loops without considering disk layouts of the arrays involved does not lengthen disk inter-access times. In other words, as mentioned earlier, loop distribution and loop tiling make sense in our context only if they are accompanied with a suitable layout optimization (DL). In comparison, five out of our six benchmark codes can achieve further energy savings from one of the LF+DL and TL+DL versions. Specifically, four benchmark programs, namely, *swim*, *mgrid*, *applu*, and *mesa*, get additional benefits from using the LF+DL version, while three benchmark programs, namely, *wupwise*, *applu*, and *mesa*, show additional benefits from the TL+DL version. The reason why different benchmark programs exhibit different trends is that each benchmark has different access patterns. For example, two benchmark programs, *wupwise* and *galgel*, do not contain any fissionable loop nests; so, we were unable to obtain any additional energy savings through loop distribution. However, *wupwise* can achieve energy savings with TL+DL because it contains an access pattern which is not conforming the data layout. So, after transforming data layout along with tiling, we achieve additional savings. Note that, *galgel* does not obtain any savings from LF+DL or TL+DL, because the loop nests it contains are not fissionable as mentioned earlier, and the access pattern it exhibits already conforms the underlying data layout. Maybe the most interesting trend that can be observed from Figure 13 is that our code transformations make the TPM strategy a viable option for this set of benchmarks. In other words, while the CMTPM strategy could not find any opportunity to save energy, the code transformations create

```
INPUT:
K loop nests and disk layout information.
OUTPUT:
An energy-optimized loop-fissioned code and the transformed disk layout.
Begin
AG ← ∅        // AG keeps array groups
for each loop nest do
    for each statement in the loop nest do
        B ← array group accessed by the statement
        if (B ∩ (all sets in AG)) == ∅ then
            Add B to AG as a new set
        else
            Union B and the overlapping set in AG
        end if
    end for
end for
Generate fissioned loops.
Allocate disks to array groups based on total data size in each group.
End
```

Figure 11. Loop fissioning algorithm.

```
INPUT:
A loop nest and disk layout information.
OUTPUT:
An energy-optimized tiled code and the transformed disk layout.
Begin
N ← the number of arrays accessed by the nest
Create tiled loop nest with tile size, TS
for i = 1 to N do
    Determine the data size, DS(i), required while iterating given TS.
end for
for each array used in the loop nest do
    if data access pattern ≠ storage pattern then
        Transform the data layout (e.g., from row-major
            to column-major).
    end if
end for
Reshape access patterns of arrays.
for i = 1 to N do
    stripe_size(i) ← DS(i) of each array
end for
End
```

Figure 12. Loop tiling algorithm.

such opportunities for it, and consequently it reduces the energy consumption of the base case by 31%, on average. Overall, these results show that layout-aware code transformations, whose further details are omitted here due to lack of space, can be useful in practice. Finally, while we focused here on two specific code transformations only, we believe that most of the other known loop transformations can also be adapted to work with disk layouts for increasing disk inter-access times.

7. Concluding Remarks

This paper presents a compiler-driven approach to disk power management for server environments that execute large, array-dominated scientific applications. As opposed to most of the previous work on the disk power management, our approach is proactive, and uses compiler analysis to identify disk idle and active times. This allows our approach to select the most appropriate power mode for a given disk at any time. In addition, the paper demonstrates that loop distribution and loop tiling can be very useful in increasing the benefits of disk power management strategies if they could be made disk layout aware. Based on our experimental evaluation, we conclude that:

- For array-intensive scientific applications, the compiler can extract disk access pattern, and use it for placing disks into the most suitable low-power modes. In

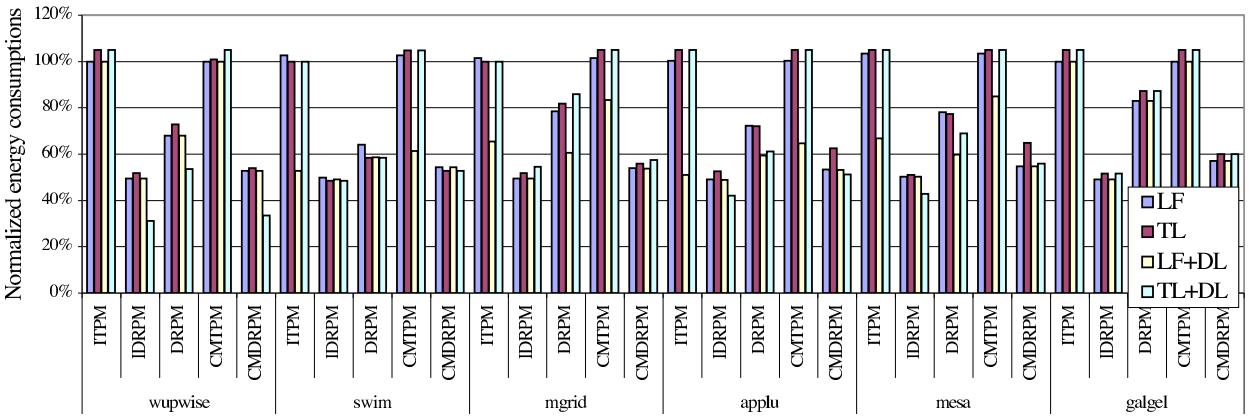

Figure 13. Normalized energy consumptions.

principle, this approach can be used with both TPM and DRPM disks.

- The compiler-directed proactive approach to disk power management is successful in improving the behavior of the DRPM based scheme. On average, it brings an additional 18% energy savings over the hardware-based DRPM.
- The effectiveness of TPM and DRPM can be increased by employing disk layout aware code transformations. Specifically, our results show that two loop restructuring techniques, namely, loop distribution and loop tiling, are very effective in some of the benchmark codes tested, and they can make TPM a serious alternative for array-based scientific codes.

References

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

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

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

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

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

[6] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. DRAM Energy Management Using Software and Hardware Directed Power Mode Control. In *Proc. of the International Conference on High Performance Computer Architecture*, pages 159–169, Jan 2001.

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

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

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

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

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

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

[13] 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. In *Proc. of International Symposium on Low Power Electronics and Design*, pages 459–464. ACM, August 2003.

[14] M. Lam, E. Rothberg, and M. Wolf. The Cache Performance of Blocked Algorithms. In *Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[15] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power Aware Page Allocation. In *Proc. of Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–116, November 2000.

[16] E. Pinheiro and R. Bianchini. Energy Conservation Techniques for Disk Array-Based Servers. In *Proc. of the 18th International Conference on Supercomputing*, pages 66–78, June 2004.

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

[18] SPEC. http://www.specbench.org/cpu2000/CFP2000/.

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