A Compiler-Guided Approach for Reducing Disk Power Consumption by Exploiting Disk Access Locality*

Seung Woo Son Guangyu Chen Mahmut Kandemir
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802, USA
{sson,gchen,kandemir}@cse.psu.edu

Abstract

Power consumption of large servers and clusters has recently been a popular research topic, since this issue is important from both technical and environmental viewpoints. The prior research proposed disk power management as one of the important ways of reducing overall power of a large system and considered both hardware-based and software-guided disk power reduction schemes. One of the common characteristics of the previously proposed approaches to disk power reduction is that they work with a given disk access pattern. In comparison, the goal of the approach proposed in this paper is to restructure application code using an optimizing compiler so that disk idle periods are lengthened. This in turn allows the underlying disk power management scheme to be more effective since such schemes usually prefer the long idle periods over the short ones. Our approach targets at large scientific applications that operate on disk-resident arrays using nested loops and exhibit regular data access patterns. To test the effectiveness of the proposed approach, we implemented it within an optimizing compiler and performed experiments with six data-intensive applications that manipulate disk-resident data. Our experimental analysis shows that the proposed approach is very successful in practice and reduces the total disk energy consumption on average by 18.17%, as compared to an execution without any disk power management, and by 11.55%, as compared to an execution that employs disks with low-power capabilities without our code restructuring approach.

1. Introduction

Power consumption of high-performance systems is becoming an increasing concern for system designers and software writers. There are several reasons for this. First, high power consumption requires sophisticated cooling techniques which can be very expensive for large computing systems [4]. Second, high power consumption brings a number of reliability related problems along with it (e.g., due to wearing out some components earlier than normal wear-out period [10]). Third, it is well known that high power consumption is harmful for the environment. Storage subsystems in particular have received significant attention lately due to their substantial contribution to overall system power. In fact, prior research [17] states that disk storage can be responsible from up to 27% of total system power consumed by data centers. Disk power consumption can be very high in systems that execute large data-intensive scientific applications, e.g., those from the domain of astrophysics, genome research, computational chemistry, and nuclear simulation.

Motivated by these observations, recent research has focused on reducing power consumption of disk storage in the context of high-performance systems. We can roughly divide the proposed approaches into two categories: hardware and software. On the hardware side, two frequently discussed techniques are traditional disk power management (spinning down an idle disk [12]) and dynamic speed setting (changing the rotation speed of disks based on the dynamic workload behavior [13]). In comparison, the prior software work [25] studied explicit management of disk power modes and automatic disk layout detection. Irrespective of whether hardware or software based, most prior techniques to disk power management become more effective with long disk idle periods (instead of short ones). For example, a longer idle period can enable us exercise a lower disk speed than a shorter idle period can do. Similarly, a sufficiently long idle period may allow us spin down the disk.

Focusing on large disk-intensive scientific applications with regular data access patterns, this paper proposes and experimentally evaluates a *compiler-based* approach to disk power management. The idea is to restructure an application code such that *disk reuse* is maximized, i.e., the application accesses as many data elements as possible from a set of disks before moving to the next set. While the application is exercising a particular set of disks controlled by an

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

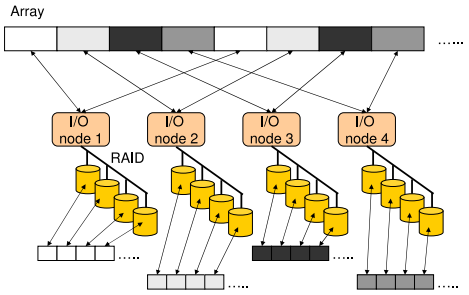


Figure 1. Two-level striping of array data across disks.

I/O node, the disks controlled by the remaining I/O nodes can be spun down for saving power or be operated under low (rotation) speeds. To enable this optimization, we propose the disk layout of array data to be *exposed* to the compiler. Using this disk layout information and the data access pattern extracted from source code, our compiler restructures data access pattern such that disk reuse is significantly improved. In this paper, we address this problem in the context of both single-CPU based execution and multi-CPU based execution, since they require different types of code transformations and generations.

To test the effectiveness of the proposed approach, we implemented it within an optimizing compiler (built upon SUIF [28]) and performed experiments with six large data-intensive applications that manipulate disk-resident data. Our experimental analysis shows that the proposed approach is very successful in practice and reduces total disk energy on average by 18.17%, as compared to an execution without any power management, and by 11.55%, as compared to an execution that employs disk with low-power capabilities without our approach. In other words, providing compiler support can bring additional savings over what could be achieved using hardware-based power management alone.

The remainder of this paper is organized as follows. The next section describes the storage subsystem we consider in this work and defines what we mean by disk layout. Section 3 discusses the related work on disk power management. Section 4 reviews two prior approaches to disk power management in detail. Sections 5 and 6 present our approaches for the single-CPU and multi-CPU cases, respectively. Section 7 discusses our experimental results, and Section 8 concludes the paper by summarizing our major findings and outlining the future research directions on this topic.

2. Storage Architecture, Disk Layouts, and Assumptions

The storage system considered in this work is shown in Figure 1 at a high level. The disk requests in this architecture are directed to I/O nodes over which the array data are striped. Each I/O node includes a CPU along with the memory components and a disk subsystem (which is typically a RAID architecture [9]). The stripes assigned to an I/O node are further striped at the RAID level (depending on the specific RAID implementation adopted). Therefore, as depicted in Figure 1, each data array in our storage architecture is striped at *two different levels* (I/O node level and RAID level). Of these, the I/O node level striping is visible to the software (to the compiler in our case) and can be controlled using calls from the underlying I/O library and/or the parallel file system used. For example, in PVFS [22], one can obtain the I/O node level striping information of files by inspecting the `pvfs_filestat` structure, which includes the stripe unit and the number of disks used for striping. The RAID level striping, however, is hidden from the software.

In this paper, the compiler manages disk power at an I/O node granularity. That is, when we talk about "spinning down a disk" or "placing a disk into the low-power mode", what we really mean "spinning down the disks controlled by an I/O node" and "placing the disks controlled by an I/O node in the low-power mode". However, for the ease of discussion, we use the term "disk" instead of "I/O node" when we discuss our approach. The "disk layout" concept used in the rest of this paper refers to the I/O node level striping; i.e., when we mention "striping", we mean the striping at the I/O node level.

In our experimental evaluation, we assume a one-to-one mapping between data arrays and files. In other words, we assume that each data array is stored in a single file and a file contains only a single array. Under this assumption, one can talk about "striping an array over the I/O nodes." While we can relax this assumption by allowing one-to-many and many-to-one mappings between the files and the data arrays, we do not evaluate these options in this paper.

We make two assumptions in this work. First, we assume that the I/O node level striping can be exposed to the compiler. This is possible because current parallel file systems and run-time libraries provide interfaces for this. Note that in this work the compiler does not modify disk layouts; it just restructures application codes based on disk layout information exposed to it and data access patterns extracted by it. Our second assumption is that the disk system is exercised by a single application at a time (of course, the different applications can use the same system at different times). Therefore, the compiler can manage/control the disk power consumption by placing unused disks (I/O nodes) in the low-power modes. Since by spinning down a disk (putting it in a low-power mode) we do *not* destroy the data itself, our approach will *not* create a correctness issue if the second assumption fails. However, in this case, our energy savings can be reduced and we can incur I/O performance degradations for some applications. We believe that the disk usage information estimated by the compiler can be passed to the operating system (OS) at specific program points, and the OS in turn can use this information to implement more global power management algorithms. However, such extensions are not the focus of this paper. Our goal instead is to evaluate the potential power savings from a single application's viewpoint.

3. Discussion of Related Work

Most of the prior studies trying to conserve disk power/energy consumption are based on exploiting disk idle periods during the course of execution. Since the power consumption of a disk in the idle state is nearly the same as that of active state when the disk services I/O requests, initial efforts concentrated on providing low-power operating modes within the disk hardware itself and studied power-saving techniques such as placing idle disks into low-power modes [12]. Note that, placing a disk into a low-power mode and switching it back to the active state can consume both extra disk I/O time and additional disk power; therefore, it is always beneficial to work with long idle periods. That is, extending idle periods can increase the effectiveness of the underlying power management mechanism.

Zhu et al and Papathanasiou et al consider power-aware caching and prefetching strategies in the storage cache memory and OS layer to increase disk idle periods [19, 29, 30]. The idea behind the energy efficient prefetching is to create burst access patterns, rather than spreading disk accesses over the entire execution time. This burst disk accesses in turn increase the idle periods of a given disk so that it can be placed into a low-power operating mode. In [29], Zhu et al study a power-aware cache replacement algorithm, called PA-LRU, in the context of large storage systems, which are typically equipped with several GBs of cache memory if aggregated across all I/O nodes. The main idea is to selectively keep cache blocks from certain disks, which exhibit longer idle periods with high probability based on dynamically traced workload data, so that the disks can be placed in low-power mode for a longer period of time. In another paper, Zhu et al [30] propose an approach, called PB-LRU (Partition-Based LRU), for the same problem. PB-LRU implements cache replacement techniques for disk arrays equipped with multi-speed disks.

The next group of studies focus on compiler-based techniques to increase idle periods of disks. Heath et al [14] study an application code transformation technique for energy/performance-aware device management by generating I/O burstiness. Son et al [25] propose a compiler-based proactive disk power management scheme for scientific applications. Since the compiler can predict disk access patterns, one can then insert explicit power management calls in the code, thereby eliminating performance penalty due to reactive disk spin-up. Son et al also describe a compiler-driven approach to reduce disk power consumption in the presence of parallel disk systems [24]. In another paper, Son et al [23] propose a compiler technique that determines energy-efficient disk layout parameters, i.e., stripe size, number of disks used for striping, and the first disk where file striping starts.

The last group of studies we discuss focus on file level granularity to conserve server disk power consumption. The previous disk power management techniques discussed so far manipulate disk data at a fine granularity (e.g., a disk block). Since large data centers host large amounts of data maintained for several application domains, they also demonstrate a locality of disk partition size, which means that not all disks need to be in active mode to service an application's I/O requests. Based on this observation, in [11], MAID (Massive Arrays of Idle Disks) is proposed for saving energy by reducing spin-ups of data drives using a small number of disks as cache drives. The cache drives act as caches for the requests to the disk array, and they allow the unused disk arrays remain in the low-power modes for longer periods of time. Pinheiro and Bianchini [20] describe a data migration technique, called PDC (Popular Data Concentration), which dynamically moves the most frequently-used disk data to a subset of the disks in the system, so that the idle disks can be placed into low-power modes.

The work described in this paper is different from all these prior studies. Instead of controlling the disk power modes by exploiting disk idle periods, we propose *compiler-directed code restructuring and generation* for increasing idle periods of disks. The advantage of doing so is that any disk power management scheme (whether hardware or software based) that exploits disk idle periods can be more effective when it is used along with our approach. Therefore, in a sense, our work is complementary to the prior research such as [12], [25], and [30]. There also exist several studies on compiler optimizations for out-of-core applications [5, 6, 7, 16, 18, 26]. The work described in this paper is different from all these, since our goal is to reduce power consumption.

4. TPM and DRPM

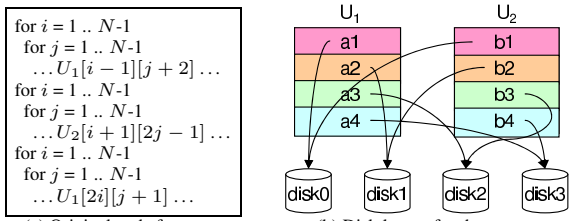
In this work, we evaluate the impact of our approach on saving disk energy under two power management mechanisms proposed in earlier research. Therefore, below, we describe these two mechanisms in more detail.

As mentioned earlier, the basic approach to reduce disk power consumption is to exploit disk idle times. That is, if a disk is idle for a certain period of time, then it can be spun down, and the disk remains in this spin-down mode until it receives a new request, at which time it starts transitioning to the full operational (active) mode. Note that, upon receiving a new I/O request, the disk must be spun up to service it. This technique, denoted as *TPM* (*traditional power management* [12]) in this paper, has been extensively studied in the context of mobile disks since such systems are operating under tight battery constraints. It should be noted that it takes a certain amount of time for a disk to spin up and down (tens of seconds in a modern server disk), and this in turn incurs certain performance penalties. Although TPM is an effective way of conserving disk power in the laptop/desktop based environments, several recent studies also show that it is not a viable option in the context of server class disks because of the following two reasons. First, the server workloads are generally small and non-contiguous, and consequently, disk idle times are not long enough to accommodate TPM. Second, server class disks are operated at a very high rotational speed to meet I/O performance re-

quired by server clusters, and the disk spin-up/down times are really long, which in turn makes it difficult to exploit the observed idle periods. Based on this argument, a dynamic speed-setting approach (referred to as *DRPM* in this paper) was proposed in the literature [13], in which the disk drive provides multiple rotation speeds (RPM levels). The disk drive can service while running at the lower RPM level, which typically consumes less energy compared to servicing I/O requests at maximum speed. An application that executes on a platform with DRPM-capable disks can choose one of the RPM levels provided, dynamically at runtime, to achieve disk energy savings without hurting execution time much. In a sense, DRPM shares similar idea with CPU voltage scaling techniques [27] proposed in the literature because the selection of the disk speed level is made based on the change in the average disk response time recorded for n-request windows. DRPM also incurs some performance penalty because servicing at a lower RPM can potentially affect the I/O response time. It has been shown that DRPM can save significant amount of disk power consumption by exploiting even small idle times, and it incurs relatively small performance penalty compared to TPM. A similar technique based on a two-speed disk architecture has been proposed and evaluated in [8]. One of the important characteristics of our scheme is that it can be used in conjunction with both TPM and DRPM and can significantly increase their effectiveness, as will be discussed later in detail. In the rest of this paper, we say that a disk is "placed into the low-power mode", when it is either spun down (in case of TPM disks) or switched to a lower speed than the maximum one (in case of DRPM disks).

5. Code Structuring for Single Processor Based Execution

To illustrate how compiler-directed code restructuring can be beneficial in reducing the disk energy consumption, we first consider the simple code fragment (written in a pseudo-language format) shown in Figure 2(a) in which three different loop nests manipulate two disk-resident arrays (U_1 and U_2) using entirely different access patterns. For illustrative purposes, let us assume that we have a total of 4 disks, and arrays U_1 and U_2 are striped over all these 4 disks, as depicted in Figure 2(b). Each disk in the system is assumed to be equipped with either TPM or DRPM. Our approach starts with the layout of the arrays on the disk, and operates as follows. Let $\mathcal{Q}_i$ denote the set of iterations of loop nest i, where $1 \leq i \leq 3$. Assume further that $\mathcal{Q} = \mathcal{Q}_1 \cup \mathcal{Q}_2 \cup \mathcal{Q}_3$, i.e., $\mathcal{Q}$ represents the set of all the iterations to be executed by this program fragment. We first determine the set of iterations from $\mathcal{Q}$ that access $disk0$, and schedule them (i.e., they become the first set of iterations to be executed by the transformed code). Note that, these iterations can come from any subset of $\mathcal{Q}_1$, $\mathcal{Q}_2$, and $\mathcal{Q}_3$. Let us refer to this set of iterations as $\mathcal{Q}_{d0}$. Subsequently, we update $\mathcal{Q} = \mathcal{Q} - \mathcal{Q}_{d0}$, and determine all the loop iterations from this set that access $disk1$ (call this set $\mathcal{Q}_{d1}$), and



(a) Original code fragment in a pseudo-code representation.

(b) Disk layout for the arrays.

for ii = 1 .. N/K
for i = max(2*K*ii-(2*K-1),1) .. min(2*K*ii, N-1)
for j = 1 .. N-1
$\ldots U_1[i-1][j+2]\ldots$
for i = max(2*K*ii-(2*K-1), 1) .. min(2*K*ii-2, N-1)
for j = 1 .. N-1
$\ldots U_2[i+1][2j-1]\ldots$
for i = max(K*ii-K, 1) .. min(K*ii-1, N-1)
for j = 1 .. N-1
$\ldots U_1[2i][j+1]\ldots$

(c) Transformed code fragment.

Figure 2. Code restructuring example. N/K is the total number of stripes and arrays U_1 and U_2 are of the same size, $2N \times 2N$.

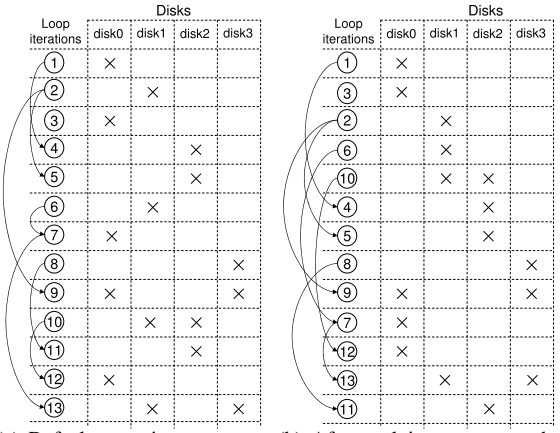
schedule them. Next, we update $\mathcal{Q} = \mathcal{Q} - \mathcal{Q}_{d1}$, and extract from this new set all the loop iterations that access $disk2$. If we use $\mathcal{Q}_{d2}$ to represent this set of iterations, we next update $\mathcal{Q} = \mathcal{Q} - \mathcal{Q}_{d2}$. The remaining iterations, if any, are the ones that exclusively access $disk3$, which constitute the last set of iterations to be scheduled. To sum up, what this approach does is to isolate, for a given period of time, accesses to a single disk as much as possible. For the code fragment shown in Figure 2(a) and the disk layouts depicted in Figure 2(b), Figure 2(c) gives the transformed code fragment. The interesting point about this code fragment is that it completes all accesses to a disk before moving to the next disk, and each disk is visited only once, i.e., it maximizes disk reuse by clustering accesses to a single disk in a given time frame.

It needs to be noted that this ideal disk access pattern (i.e., perfect disk reuse) may not be possible in general. This can be so because of two potential reasons. First, in many loop nests, we may have multiple array references. Consequently, a given loop iteration can access different array elements that reside in different disks. Second, the data dependences between the loop iterations can prevent the iteration reordering required by this approach. However, our transformation strategy that targets disk reuse can still be expected to be beneficial even in such cases since it can achieve a certain level of clustering, as will be demonstrated by our experimental analysis presented later. In other words, even if it may not be possible to achieve perfect disk reuse (where each disk is visited only once), our approach can still improve disk access locality, which in turn can translate to energy savings.

Figure 3 gives the pseudo code of our disk reuse oriented code restructuring algorithm for the single processor based execution case. We use the Omega library [21], a polyhe-

INPUT:
N Loop nests and disk layout information.
OUTPUT:
An energy-optimized restructured code.
Begin
$stripe\_factor \leftarrow$ the number of disks.
$\mathcal{Q}_i \leftarrow$ the sets of iterations of loop nest i.
$\mathcal{Q} \leftarrow \mathcal{Q}_1 \cup \mathcal{Q}_2 \cup \cdots \cup \mathcal{Q}_N$.
while $\mathcal{Q} \neq \emptyset$ **do**
  for $i = 0$ **to** $stripe\_factor - 1$ **do**
    $\mathcal{Q}_{di} \leftarrow$ all schedulable loop iterations accessing disk(i) from $\mathcal{Q}$.
    call Omega_lib ($\mathcal{Q}_{di}$).
    $\mathcal{Q} \leftarrow \mathcal{Q} - \mathcal{Q}_{di}$.
  end for
end while
Form a merged loop nest using an outermost loop iterator with iteration count of $stripe\_factor$.
End

Figure 3. Sketch of our code restructuring algorithm. If the code does not have any data dependence, the while-loop in the algorithm iterates only once. Omega_lib generates the loop nests that iterate over the data elements in $\mathcal{Q}_{di}$ using the *codegen* utility.



(a) Default execution sequence. (b) After applying our approach.

Figure 4. An example that illustrates how our code restructuring algorithm works with loop iterations having data dependences (captured by arrows). The symbol $\times$ in a cell represents the accessed disk while executing the corresponding iteration.

dral tool, to produce the restructured loop nests. Omega library allows user define Presburger formulas that involve arithmetic and logic connectives and existential ($\exists$) and universal ($\forall$) quantifiers. In our context, this library is used to build loop nests that enumerate iterations in sets $\mathcal{Q}_{di}$. The algorithm given in Figure 3 considers the cases where the code has data dependences. If, on the other hand, there is no dependence in the code, all loop iterations that access a given disk can be scheduled successively. In contrast, if there exists dependences in the code, the algorithm can only schedule (when considering a disk) the some portion of iterations that can be executed just before the dependence occurs (i.e., in this case, we may need to visit a disk more than once). Since not all of loop iterations are scheduled at first run over stripe factor, $\mathcal{Q}$ is not empty. Therefore, the remaining loop iterations will be scheduled until all iterations are scheduled, i.e., until $\mathcal{Q}$ is empty (which is captured by the while-loop).

We now want to discuss how our approach works using the example given in Figure 4. In this example, for illustrative purposes, we assume that there are 13 loop iterations and they access 4 disks during execution. Figure 4(a) shows how these iterations access the disks (i.e., the default access sequence). In this default execution (i.e., without any restructuring), the execution sequence is $1\rightarrow 2\rightarrow 3\rightarrow \ldots \rightarrow 13$. If we apply the algorithm in Figure 3, on the other hand, we start by scheduling loop iterations accessing $disk0$, which gives us the sequence of $1\rightarrow 3$. Since there are data dependences from iterations 2, 6, and 10 to iterations 9, 7, and 12, we move to iterations accessing $disk1$, instead of scheduling iterations 9, 7, and 12 at this point. In processing $disk1$, we schedule iterations 2, 6, and 10. After that, we focus on $disk2$ and $disk3$, and schedule iterations 4, 5, 8, and 9. This completes the first iteration of the while-loop in our algorithm in Figure 3. In the next iteration of the while-loop, we first focus on $disk0$ again, and schedule iterations 7 and 12, and so on. As a result, the new execution sequence is generated as shown in Figure 4(b). This new execution sequence clearly increases the length of the idle periods for each disk, thereby increasing opportunities for saving energy.

6. Code Structuring for Multiprocessor Based Execution

While the code restructuring approach presented in the previous section increases disk reuse through clustering, it may not work very well when multiple processors are used for executing a given parallel application. This is because in this case the disk requests coming from different processors may interleave in time, and this in turn, reduces the disk idle periods, thereby, cutting down the potential energy savings. In this section, we discuss our code restructuring for the multi-processor execution case. First, in Section 6.1, we revisit the conventional loop based parallelization theory, and then, in Section 6.2, we present the details of our proposed parallelization scheme, oriented toward maximizing disk idle periods during multi-processor execution.

6.1 Background: Loop Based Code Parallelization

The loop based code parallelization focuses on a single loop nest at a time, and parallelizes it using the data dependence information extracted by the compiler. While several proposals exist in the compiler literature (e.g., [2]), the main goal behind all these techniques is to rewrite a given loop nest in a form that allows parallel execution of independent loop iterations. To minimize synchronization costs, it is also important that we obtain coarse grain parallelism, as opposed to fine grain parallelism. In terms of parallel ex-

ecution, this means parallelizing the *outermost* (parallelizable) loop as much as possible for each nest.

Each execution of a loop nest body can be represented by an *iteration vector,* each entry of which corresponds to a loop, starting from the top. When there is no confusion, we use the terms "loop iteration" and "iteration vector" interchangeably. An iteration vector represents the executions of all the statements in the loop body (under the specified values of the iterators in the vector).

If a loop iteration $\vec{q_2}$ depends on an iteration $\vec{q_1}$ (where $\vec{q_2} > \vec{q_1}$, the difference between them, $\vec{q_2} - \vec{q_1}$ is called the *data dependence vector* [3]. Note that, in this work, we are mainly interested in data dependences. This is because the application codes we consider are loop nest intensive (i.e., operate on disk-resident arrays using nested loops) and they do not contain conditional flow of executions. We are mostly interested in cases where all the entries of a dependence vector are constants, in which case it is also referred to as the *distance vector* [3]. The distance vectors extracted from a loop nest collectively define a *distance matrix,* whose rows are made of distance vectors. Let us focus on an arbitrary distance vector $\vec{d}$ extracted from a nest with n loops:

$$\vec{d} = (d_1 \quad d_2 \quad d_3 \quad \cdots \quad d_{n-1} \quad d_n)^T.$$

Considering (only) this distance vector, the kth loop can be parallelized if at least one of the two conditions below are satisfied [3]:

- $d_k = 0$, or
- $(d_1 \quad d_2 \quad \cdots \quad d_{k-1})^T$ is lexicographically positive.

We say that vector $\vec{d} = (d_1 d_2 \cdots d_n)$ is lexicographically less than (shown as $<$) vector $\vec{d'} = (d'_1 d'_2 \cdots d'_n)$ if there is a c such that $1 \leq c \leq n$ and $d_i = d'_i$ for all $i < c$ and $d_c < d'_c$. A vector is said to be lexicographically positive (negative) if it is greater than (less than) the zero vector. In obtaining the coarsest grain parallelism, the compiler normally parallelizes only the kth loop such that this loop parallelizable and none of the loops from top down to the $(k-1)$th loop is parallelizable. If there are multiple distance vectors in the nest, a loop is parallelizable if and only if it is parallelizable according to all these distance vectors. The different approaches to coarse grain loop based parallelization differ mostly in their capability of extracting the highest level of parallelism in a given loop nest.

One of the serious drawbacks of loop based parallelization is that it does not capture the data sharings between the different loop nests, which can be a significant problem as far as disk reuse is concerned. This is illustrated in Figure 5, which depicts how an example application with three separate loop nests accesses a two-dimensional array. The left part of the figure shows the iteration spaces of the nests. Each iteration space is assumed to be divided into 4 parts as a result of parallelization over 4 processors. That is, each processor is set to execute one fourth of the original

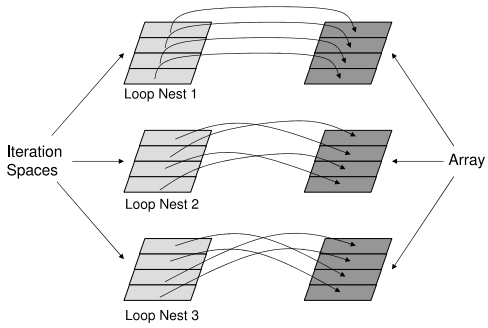
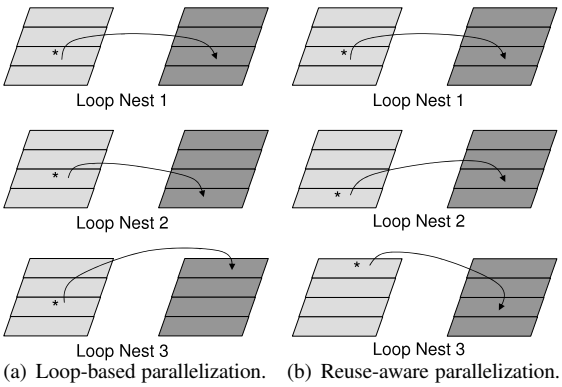


Figure 5. An example data access pattern scenario that involves four processors. In this scenario, three different loop nests access the same array.



(a) Loop-based parallelization. (b) Reuse-aware parallelization.

Figure 6. Two different parallelizations from the perspective of a given processor.

iteration space of each nest. The array (data space) manipulated by these loop nests (note that all three processors manipulate the same array) is also shown as divided into four regions (on the right of the figure). The arrows from the iteration spaces to the array space indicate which array region each part of the iteration space accesses. Since the loop based parallelization does not capture the data sharings between the different loop nests, it can assign to a processor from each nest the (iteration space) part in the same position. As a result, a given processor can have the access pattern shown in Figure 6(a). Let us focus on the portions of the iteration spaces marked "*", which are assigned to the same processor under the loop based scheme. The problem with this access pattern is that, at each nest, the processor in question accesses a different data region of the array. Therefore, one would not expect a good disk reuse from this data access pattern. The objective of the disk layout-aware (reuse-aware) code parallelization scheme discussed in the next subsection is to address this problem.

6.2. Disk Layout-Aware Code Parallelization

6.2.1 High-Level View of Our Approach

Figure 6(b) illustrates, through an example, our approach to disk reuse-aware code parallelization. The figure shows the

assigned parts from each loop nest to a particular processor. The portions marked "*" indicate the iterations assigned to the same processor. This reuse-aware assignment differs from the one shown in Figure 6(a) in two ways. First, the same processor is assigned to different parts in the different iteration spaces (i.e., not the corresponding parts). Second, and more importantly, the same processor accesses the same array region in each nest, which means that one can expect a good disk reuse. The main goal of the disk layout aware strategy is to achieve the highest disk reuse possible. Clearly, this ideal scenario (as depicted in Figure 6(b)) may not be achieved in all cases, due to the data dependences between the different loop iterations. However, our approach tries to exploit the maximum possible disk reuse allowed by data dependences. The next subsection explains the mathematical engine behind this code parallelization scheme. It needs to be emphasized that this parallelization scheme in a sense partitions the disks in the storage system across the processors by localizing accesses to each disk to a single processor as much as possible. After this parallelization, the disk reuse based restructuring (explained in Section 5) can be applied to the code of each processor separately to further increase disk idleness (and energy savings).

6.2.2 Mathematical Details

We now discuss the details of the mathematical engine behind our approach to disk reuse in the multiprocessor case. Our approach is data space oriented, meaning that it decides the set of loop iterations to be assigned to each processor considering the arrays accessed by the application. We use $\mathcal{Q}_k$ (where $1 \leq k \leq n$) to denote the set of iterations that will be executed by loop nest k. Let us focus on an array U_j (where $1 \leq j \leq m$) manipulated by the application. We use $\mathcal{Z}_j$ to represent the set of data items in U_j. Note that $\mathcal{Z}_j$ defines a rectilinear polyhedron. We assume that there are p processors in the system over which the application code is to be parallelized.

In the first step of our approach, we logically divide the array into p regions and each region is assigned to a processor (note that the data in each region can span multiple disks). We now focus on processor s (where $1 \leq s \leq p$) and loop nest k (where $1 \leq k \leq n$). Let $\mathcal{Z}_{s,j}$ be the data elements from array U_j that are assigned to processor s (we will discuss shortly how this data assignment is actually made). We use $\mathcal{Q}_{s,j,k}$ to represent the set of loop iterations from loop nest k that touch the elements in $\mathcal{Z}_{s,j}$. Our parallelization strategy assigns the iterations in $\mathcal{Q}_{s,j,k}$ to processor s. In other words, each processor executes the loop iterations that access the array region it is assigned to. This iteration assignment can be repeated for each loop nest. In other words, processor s is assigned iterations $\mathcal{Q}_{s,j,1}$, $\mathcal{Q}_{s,j,2}$, ..., $\mathcal{Q}_{s,j,n}$. The common characteristic of these sets is that all the iterations in them access $\mathcal{Z}_{s,j}$. Consequently, when all the iteration assignments (for all loop nests) are complete, we have the scenario shown in Figure 6(b) for the example in Figure 5.

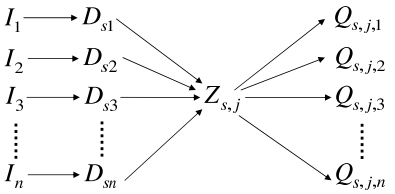


Figure 7. The process of determining the set of iterations (from all the nests) that will be executed by processor s (assuming that we have n nests).

At this point, there are three important issues that need to be addressed. The first issue is the problem of (logically) dividing the array elements across the processors. The second one is regarding the fact that not all the loop nests access all the elements of a given disk-resident array. Consequently, we need a strategy to handle the case when one or more loop nests access only a portion of the array. The third issue is that normally an application processes multiple disk-resident arrays and our iteration mapping must be carried out considering all the arrays. Otherwise (i.e., if we consider only one array), the resulting parallelized code may not be able to exploit disk reuse for the arrays not considered during the workload assignment. In the following paragraphs, we elaborate on these three issues.

Dividing array elements across processors is very important as it determines the data (disk) access pattern and thus influences parallelism and disk reuse. Our approach to this problem can be explained as follows. For each loop nest, we extract the maximum parallelism using a previously published approach in the literature [1]. This approach implements a method for deriving an optimal hyperparallelepiped tiling of iteration space for minimal communication in multiprocessors. It uses the notion of *uniformly intersecting references* to capture data reuse among array references and estimates interprocessor data communication traffic based on a data footprint concept. After parallelizing the code using the approach in [1], for each loop nest, our approach determines the set of data elements accessed by each processor s. In determining this set of elements, we build the following set, assuming that $\mathcal{I}_s$ is the set of iterations assigned to processor s (by the loop parallelization used) and $\mathcal{D}_s$ is the set of array elements we want to determine:

$$\mathcal{D}_s \quad = \quad \{\vec{d} \mid \exists \vec{I} \in \mathcal{I}_s, \ \ \exists R \in \mathcal{R}_s \text{ such that } R(\vec{I}) = \vec{d}\}.$$

In this formulation, $\mathcal{R}_s$ is the set of references to the disk-resident array and R represents a reference in the loop nest (i.e., a mapping from the iteration space to the data space). Since the access pattern imposed by each loop nest (on the array) can be different from the other loop nests, we next employ a *unification step* that comes up with a globally acceptable array partitioning (data mapping). This data mapping is then used for distributing the loop iterations across the processors. While it is possible to implement different unification schemes, the scheme used in this study is

a simple one that selects the most frequently requested data mapping (when all the loop nests in the application are considered). As an example of our array partitioning approach, let us assume that an array is accessed by three different loop nests. The first and the third loop nests require the array elements to be assigned to the processors in a row-block fashion (i.e., each processor is given a consecutive set of rows), whereas the second loop nest demands a column-block distribution across the processors. Let us use $\mathcal{D}_{s_1}$, $\mathcal{D}_{s_2}$ and $\mathcal{D}_{s_3}$ to denote the distributions demanded by the loop nests, i.e., row-block, column-block, and row-block in that order, from the perspective of processor s. Considering these, our approach selects the row-block distribution ($\mathcal{D}_{s_1}$), as it is requested by a larger number of processors; i.e., we set $\mathcal{Z}_{s,j}$ to $\mathcal{D}_{s_1}$. Figure 7 summarizes the process of determining the loop iterations that will be executed by processor s. Basically, using the approach in [1], we first determine the sets $\mathcal{D}_{s_1}$, $\mathcal{D}_{s_2}$, ..., $\mathcal{D}_{s_n}$. We next obtain $\mathcal{Z}_{s,j}$ using the strategy explained in the previous paragraph, and then determine $\mathcal{Q}_{s,j,1}$, $\mathcal{Q}_{s,j,2}$, ..., $\mathcal{Q}_{s,j,n}$, i.e., the iterations from the different nests that are assigned to processor s.

For the second issue, let us consider two loop nests, k and l, that access the same array (U_j). We use $\mathcal{M'}_{j,k}$ and $\mathcal{M'}_{j,l}$ to denote the set of elements (of the disk-resident array U_j) accessed by nests k and l, respectively (note that $\mathcal{M'}_{j,k} \subseteq \mathcal{Z}_j$ and $\mathcal{M'}_{j,l} \subseteq \mathcal{Z}_j$). We can express the problem as follows: Divide the iterations in $\mathcal{Q}_k$ and $\mathcal{Q}_l$ across the p processors such that the parts assigned to processor s from these two nests, namely, $\mathcal{Q}_{s,j,k}$ and $\mathcal{Q}_{s,j,l}$, access the same set of elements as much as possible. In mathematical terms, our approach proceeds as follows. We first determine the set of common elements between $\mathcal{M'}_{j,k}$ and $\mathcal{M'}_{j,l}$, i.e., $M'_{j,common} = \mathcal{M'}_{j,k} \cap \mathcal{M'}_{j,l}$. Then, we assign the first $|\mathcal{M'}_{j,k}|/p$ of these ($|M'_{j,common}|$) elements to the first processor, the next $|\mathcal{M'}_{j,k}|/p$ to the second processor, and so on. At the point where we have assigned all ($|M'_{j,common}|$) elements, the remaining elements (i.e., $|\mathcal{M'}_{j,k}| - |M'_{j,common}|$) are assigned to the remaining processors. A similar process is repeated for the second loop nest (l) as well. However, in processing this loop nest, we are careful in assigning the same set of (common) elements to the same processor as in the previous loop nest. Then, based on these data assignments, we perform the iteration assignment as explained earlier in this subsection.

Our approach to the third issue – multiple disk-resident arrays accessed by the nests – can be explained as follows. We first identify affinity among the elements of the different arrays. Two array elements are said to have *affinity* if they are accessed by the same loop iteration. As an example, consider the following loop nest written in a pseudo language, and the three references (to disk-resident arrays) that appear in it:

```
for i = 1 .. M − 2
  for j = 4 .. N
    ... U1[i][j] ... U2[j][i] ... U3[i + 2][j − 3] ...
```

We note that, for a given loop iteration (a, b) in this nest,

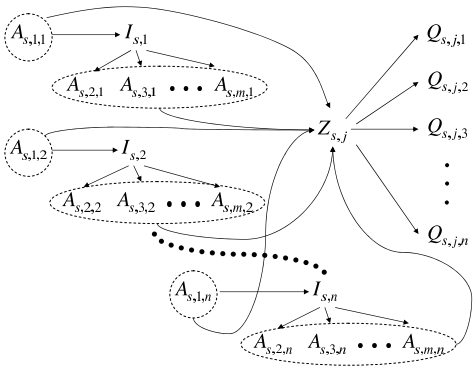


Figure 8. The process of determining the set of iterations (from all the nests) that will be executed by processor s (assuming that we have n nests).

i.e., when $i = a$ and $j = b$, the loop accesses array elements $U_1[a][b]$, $U_2[b][a]$, and $U_3[a+2][b-3]$, and thus, these three array elements have affinity. Then, instead of dividing an array into regions (as in the single array case), we divide data elements into *affinity classes*. Each affinity class contains data elements that exhibit affinity among them. Then, the iteration mapping (assignment) is carried out based on these affinity classes. In mathematical terms, let $\mathcal{A}_{s,1,k}$, $\mathcal{A}_{s,2,k}$, $\mathcal{A}_{s,3,k}$, ..., $\mathcal{A}_{s,m,k}$ be the array regions accessed by processor s from the disk-resident arrays U_1, U_2, U_3, ..., U_m, respectively, in loop nest k (i.e., they form an affinity class for processor s). In computing $\mathcal{Z}_{s,j}$, our approach uses these regions. After computing $\mathcal{Z}_{s,j}$, the sets $\mathcal{Q}_{s,j,1}$, $\mathcal{Q}_{s,j,2}$, ..., $\mathcal{Q}_{s,j,n}$, i.e., the iterations from the different nests that are assigned to processor s, can be computed as has been discussed earlier. Figure 8 gives an illustration of this process. Note that, this calculation is just for processor s and needs to be carried out for each processor separately. We also want to emphasize that the proposed approach is different from loop fusion and similar techniques that operate neighboring loop nests in the code. In contrast to these techniques, the approach proposed in this paper considers *all* the loop nests at the same time, and in general, the output (restructured) code generated by our approach cannot be obtained by simple loop fusioning.

7. Experimental Evaluation

7.1. Setup

To generate disk access patterns for our application programs, we implemented a trace generator. The cycle estimates for the loop nests were obtained from the actual execution of the programs on a SUN Blade1000 machine (UltraSPARC-III architecture operating at 750 MHz with Solaris 2.9) and these estimates were used in all our simulations. Access to disk-resident data is made at a page block granularity. In addition to the I/O trace file, our simulator needs the disk striping information such as stripe unit size, striping factor (the number of disks), and starting iodevice (disk). Using these disk parameters, the simulator deter-

mines which I/O nodes it should access when it reads an I/O request. In all the experiments reported, each I/O node has one disk and no further striping is applied at the I/O node level, i.e., the data is only striped across the I/O nodes (though the experiments with low-level striping generated similar results). In our simulator, the striping information is provided in an external file along with other simulation parameters. The default simulation parameters used in our experiments are given in Table 1.

The simulator we wrote 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 following four parameters:

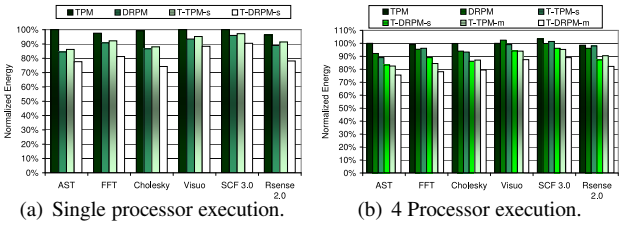
- Request arrival time: Time in milliseconds specifying the time the request arrives.
- Start block number: An integer specifying a logical disk block striped over several I/O nodes.
- Request size: An integer in bytes specifying the size of a request.
- Request type: A character specifying whether the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

Given an I/O trace file, the simulator generates statistical data for performance (the disk I/O time) and disk energy consumption. Both performance and energy statistics were calculated based on the figures extracted from the data-sheet of the IBM Ultrastar 36Z15 [15], and are given in Table 1. The values for power mode transitions are also included in Table 1. As to the power model of DRPM disks, we obtained these values using quadratic estimation described in [13].

Table 2 gives the important characteristics of the set of array-based application codes used in this study. These applications are large disk-intensive scientific codes collected from different sources. The first column in Table 2 gives the application name and the second column gives a brief description of it. The third column shows the amount of disk data manipulated by each application. The next column gives the number of disk requests made from each application. Finally, the last two columns give the disk energy consumption and disk I/O time, respectively, for each application when *no* disk power management is employed. The energy and performance numbers presented in the rest of this paper are *normalized* with respect to the values listed in these last two columns of Table 2. We also want to mention that, according to our experiments, these applications spend 75%–82% of their execution time in disk I/O.

For each application in our experimental suite, we performed experiments with seven different versions of it, which can be summarized as follows:

- Base: This is the base version that does not employ any power management method. All the reported disk energy and performance numbers presented later are given as values normalized with respect to this version, which are given in the last two columns of Table 2.
- TPM: This is the traditional disk power management strategy used in studies such as [12]. In this approach, a disk is spun down after some idleness to save power, and it is spun up when a new request arrives. Since the performance cost of spinning up is typically large, TPM can incur significant performance degradations. Also, in order for this scheme to save power, the idleness should be large enough to compensate for the spin-up and spin-down costs.
- DRPM: This is the dynamic speed-setting based strategy proposed in [13]. Considering the predicted length of the idleness, it sets the rotation speed of the disk to an appropriate level to save power. Therefore, it can save power even if the idle periods are short. The RPM level used is selected based on the degree of the response time variation, and we may incur a performance penalty.
- T-TPM-s: This corresponds to our disk reuse-based single-processor approach when it is used with TPM. As has been discussed earlier in Section 5, the compiler restructures code considering disk layout information.
- T-DRPM-s: This corresponds to our reuse-based single-processor approach when it is used with DRPM. As in the case of T-TPM-s, this version exploits disk layout information to restructure the given application code (see Section 5).
- T-TPM-m: This is similar to T-TPM-s, except that it restructures the code for multiple processors simultaneously, using the approach explained in Section 6.
- T-DRPM-m: This is similar to T-DRPM-s, except that it restructures the code for multiple processors simultaneously, using the approach explained in Section 6.



(a) Single processor execution. (b) 4 Processor execution.

Figure 9. Normalized energy consumption results.

7.2. Results

The graph in Figure 9(a) gives the energy consumption of our applications under the different schemes discussed earlier when we execute them on a single processor. One can make several observations from these results. First, the TPM version does not generate any significant savings for our applications. This is not surprising given the fact that disk idle times in these applications are not very large. In comparison, the DRPM version generates better results, achieving an average disk energy saving of 9.95%. These

Table 1. Default simulation parameters. All the arrays use the same striping (i.e., the same layout).

Disk Parameters for TPM and DRPM disks		Disk Energy Model		DRPM-specific Parameters and Striping Information	
Parameter	**Value**	**Parameter**	**Value**	**Parameter**	**Value**
Disk Model	IBM Ultrastar 36Z15	Power (active)	13.5 W	Maximum RPM Level	15,000 RPM
Interface	SCSI	Power (idle)	10.2	Minimum RPM Level	3,000 RPM
Storage Capacity	36.7 GB	Power (standby)	2.5 W	RPM Step-Size	3,000 RPM
Disk Cache Size	4 MB	Energy (spin down: idle → standby)	13 J	Window Size	100
RPM	15,000	Time (spin down: idle → standby)	1.5 sec		
Average Seek Time	3.4 ms	Energy (spin up: standby → active)	135 J	Stripe unit (stripe size)	32 KB
Average Rotation Time	2 ms	Time (spin up: standby → active)	10.9 sec	Stripe factor (number of disks)	8
Internal Transfer Rate	55 MB/sec	TPM Break-even Threshold	15.2 sec	Starting iodevice (starting disk)	1 (the first disk)

Table 2. Applications and their characteristics.

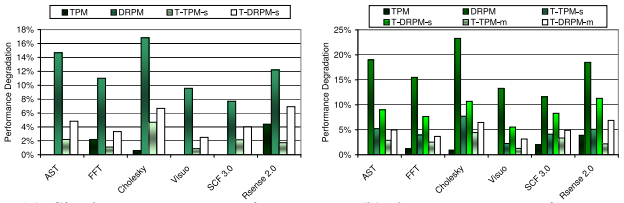
Name	Description	Data Size (GB)	Number of Disk Reqs	Base Energy (J)	I/O Time (ms)
AST	Astrophysics	153.3	148,526	44,581.1	476,278.6
FFT	Fast Fourier Transform	96.6	81,027	24,570.3	371,483.1
Cholesky	Cholesky Factorization	87.4	74,441	20,996.3	337,028.0
Visuo	3D Visualization	95.5	86,309	26,711.4	369,649.5
SCF 3.0	Quantum Chemistry	106.1	119,862	36,924.7	424,118.7
RSense 2.0	Remote Sensing Database	104.0	126,990	37,508.2	419,973.5

results are consistent with those found by the prior research [13], where they injected synthetic disk access patterns that mimic the behavior of scientific applications. Since DRPM can operate with reduced rotation speeds, it is more successful in taking advantage of short idle periods. Our next observation is that the T-TPM-s version generates much better results than TPM, and in fact, it comes close to DRPM for all the application codes tested (with an average disk energy saving of 8.30%). This result indicates that restructuring code can be very effective in increasing the benefits of the underlying power management scheme and our approach makes traditional power management a serious alternative for scientific applications. Finally, we see that the highest energy savings are obtained with the T-DRPM-s version. Specifically, this version saves 18.30% disk energy over the base scheme.

Figure 9(b) presents the energy results with the 4 processor case. As mentioned earlier, to obtain our versions running under multiple processors, we parallelized each loop nest to obtain outermost loop parallelism as much as possible. Our observations about the TPM and DRPM made above for the single processor case hold here for the 4 processor case as well. However, we see a reduction in the effectiveness of the DRPM version as interleaving disk accesses coming from multiple processors makes predicting idleness more difficult. Due to the same reason, we witness similar reduction in energy savings of T-TPM-s and T-DRPM-s as well. Specifically, the two schemes achieve 3.84% and 10.66% energy savings on average. When we look at the results of the last two versions (T-TPM-m and T-DRPM-m), on the other hand, we see that these versions bring significant benefits over T-TPM-s and T-DRPM-s. In fact, their savings are 11.04% and 18.04%, when averaged over the six applications in our experimental suite. Therefore, the most important conclusion from these results is that, in a multi-processor execution, it is not sufficient to exploit disk reuse from each processor's perspective independently. Instead, one needs to take a global approach which considers disk access patterns of all processors simultaneously. This is what our disk reuse aware parallelization approach does.

Having presented our energy results, we now turn our attention to performance results of the different versions. The performance results for the single processor execution are presented in Figure 10(a). Each bar in this figure represents the performance overhead introduced by the corresponding scheme over the base version (with respect to the last two columns of Table 2). We first observe that the TPM version does not incur significant performance penalties, mainly because it is not applicable to most of idle periods since they are very short. The DRPM scheme on the other hand incurs significant performance degradations (11.9% on the average). This is due to its inability to predict the most appropriate rotation speed for each idle period. The performance overheads caused by our two schemes, T-TPM-s and T-DRPM-s, on the other hand, are lower than DRPM since they are able to increase the length of idle periods, which in turn helps reduce the number of switches between the spin-downs and spin-ups (if TPM is used) and those between the different rotation speeds (if DRPM is employed). Specifically, the average performance degradations (i.e., increase in disk I/O time) caused by T-TPM-s and T-DRPM-s are 2.1% and 4.7%, respectively.

Let us now look at the performance degradation results when the applications are executed using four processors (see Figure 10(b)). As in the single processor based execution case, we see that the TPM version does not result in too much performance overhead (due to lack of applicability). In comparison, we see certain increase in performance degradations caused by the schemes DRPM, T-TPM-s, and T-DRPM-s (16.8%, 4.7%, and 8.7%). This is mainly due to the interleaving disk accesses coming from multiple processors. The performance overheads introduced by the T-TPM-m and T-DRPM-m however are about 2.8% and 5.0%, respectively, indicating that, in the multiprocessor case, T-TPM-m (resp. T-DRPM-m) is preferable over T-TPM-s (resp. T-DRPM-s) from the performance angle as well. Overall, when we put the results presented in bar-



(a) Single processor execution. (b) 4 processor execution.

Figure 10. Performance degradation results.

charts in Figures 9(a), 9(b), 10(a), and 10(b) together, we see that, while T-TPM-s and T-DRPM-s are clearly superior to TPM and DRPM respectively; when we move to multiprocessor-based execution, T-TPM-m and T-DRPM-m are the best choices from both the energy consumption and performance perspectives.

8. Concluding Remarks and Future Work

Disk power management has been identified as one of the major ways of saving energy in cluster systems that process data-intensive applications. While the prior research investigated techniques such as spinning-down a disk or rotating disks at a lower speed to save disk energy, most of these efforts do not make any use of the high-level data access pattern information that could be extracted from the application code by an optimizing compiler. This paper proposes and experimentally evaluates a compiler-guided approach to disk power management in parallel I/O node based systems that execute array-intensive scientific applications. The idea is to let an optimizing compiler analyze the source code and extract data access pattern, use this information along with disk layout of data to determine disk access pattern, and restructure the application code such that the disk accesses are clustered in a small set of disks at any given time; the remaining disks can then be placed into a low-power mode using any existing disk power management technique such as TPM or DRPM. We implemented this approach for both single-processor and multi-processor cases, and performed experiments with six array-intensive scientific applications that manipulate disk-resident data sets. The experimental results clearly show that the proposed approach increases the effectiveness of the two previously-proposed disk power management schemes significantly. We plan to extend this work by investigating a framework that combines application code restructuring with disk layout reorganization under a unified optimizer.

References

[1] A. Agarwal, D. Kranz, and V. Natarajan. Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared Memory Multiprocessors. In *Proceedings of International Conference on Parallel Processing*, 1993.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.

[3] U. K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, 1998.

[4] R. Bianchini and R. Rajamony. Power and Energy Management for Server Systems. Technical Report Technical Report DCS-TR-528, June 2003.

[5] R. Bordawekar, A. Choudhary, and J. Ramanujam. Automatic Optimization of Communication in Compiling Out-of-Core Stencil Codes. In *Proceedings of the 10th International Conference on Supercomputing*, pages 366–373, May 1996.

[6] A. D. Brown and T. C. Mowry. Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 31–44, October 2000.

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

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

[9] 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*, 26(2):145–185, 1994.

[10] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam. Managing Server Energy and Operational Costs in Hosting Centers. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 303–314, June 2005.

[11] D. Colarelli and D. Grunwald. Massive Arrays of Idle Disks for Storage Archives. In *Proceedings of the ACM/IEEE conference on Supercomputing*, July 2002.

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

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

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

[15] IBM. Ultrastar 36Z15 hard disk drive. `http://www.hitachigst.com/hdd/ultra/ul36z15.htm`, 2001.

[16] M. T. Kandemir, A. N. Choudhary, J. Rajanujam, and M. A. Kandaswamy. A Unified Compiler Algorithm for Optimizing Locality, Parallelism and Communication in Out-of-core Computations. In *Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems*, pages 79–92, November 1997.

[17] Maximum Institution Inc. Power, heat, and sledgehammer. `http://www.max-t.com/downloads/whitepapers/SledgehammerPowerHeat20411.pdf`, 2002.

[18] M. Paleezny, K. Kennedy, and C. Koelbel. Compiler Support for Out-of-Core Arrays on Parallel Machines. Technical Report CRPC Technical Report 94509-S, Rice University, Houston, TX, December 1994.

[19] A. E. Papathanasiou and M. L. Scott. Energy Efficient Prefetching and Caching. In *USENIX Annual Technical Conference*, pages 255–268, 2004.

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

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

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

[23] S. W. Son, G. Chen, and M. Kandemir. Disk Layout Optimization for Reducing Energy Consumption. In *Proceedings the 19th ACM International Conference on Supercomputing*, pages 274–283, June 2005.

[24] 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 *Proceedings ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 174–185, June 2005.

[25] S. W. Son, M. Kandemir, and A. Choudhary. Software-Directed Disk Power Management for Scientific Applications. In *Proceedings 19th International Parallel and Distributed Processing Symposium*, pages 4b–4b, April 2005.

[26] R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and Runtime Support for Out-of-Core HPF Programs. In *Proceedings of the 8th International Conference on Supercomputing*, pages 382–391, July 1994.

[27] M. Weiser, A. Demers, B. Welch, and S. Shenker. Scheduling for Reduced CPU Energy. In *Proceedings of Symposium on Operating System Design and Implement ation*, pages 13–23, November 1994.

[28] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J.-A. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

[29] 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 *Proceedings of the 10th International Conference on High-Performance Computer Architecture*, pages 118–129, 2004.

[30] Q. Zhu, A. Shankar, and Y. Zhou. PB-LRU: A Self-Tuning Power Aware Storage Cache Replacement Algorithm for Conserving Disk Energy. In *Proceedings of the 18th Annual International Conference on Supercomputing*, pages 79–88, 2004.