SOFTWARE-BASED DISK POWER MANAGEMENT FOR
SCIENTIFIC APPLICATIONS

A Dissertation in
Computer Science and Engineering
by
Seung Woo Son

© 2008 Seung Woo Son

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2008
The dissertation of Seung Woo Son was reviewed and approved* by the following:

Mahmut Kandemir  
Associate Professor of Computer Science and Engineering  
Graduate Officer for Computer Science and Engineering  
Dissertation Advisor  
Chair of Committee  

Padma Raghavan  
Professor of Computer Science and Engineering  

Bhavan Urgaonkar  
Assistant Professor of Computer Science and Engineering  

Long-Qing Chen  
Professor of Materials Science and Engineering  

*Signatures are on file in the Graduate School.
Abstract

Power consumption by high-performance systems is becoming an increasing concern for system designers and software writers alike. Disk subsystem is known to be a major contributor to the overall power budget of high-performance systems. Most scientific applications today rely heavily on disk I/O for out-of-core computations, checkpointing, and data visualization. To reduce excess energy consumption on disk system, prior studies proposed several hardware or OS-based disk power management schemes. While such schemes have been known to be effective in certain cases, they might miss opportunities for better energy savings due to their reactive nature. While compiler based schemes can make more accurate decisions on a given application by extracting disk access patterns statically, the lack of runtime information on the status of shared disks may lead to wrong decisions when multiple applications exercise the same set of disks concurrently. Therefore, quantitative comparison of reactive, compiler-based, and hybrid schemes is very important.

This dissertation makes four major contributions towards more effective disk power management for scientific applications that use disk-resident data frequently (i.e., so called I/O-intensive applications). First, it shows that, while conventional hardware based disk power management scheme is useful in certain cases, compiler-driven approach can be more effective for array-based scientific applications executing on parallel architectures. Second, it shows that restructuring the application code increases length of disk idle periods, thereby leading to better exploitation of available power-saving capabilities. Third, it proposes a compiler-directed energy-aware prefetching scheme for scientific applications that process disk-resident data sets. Finally, it proposes a runtime system support for software-based disk power management scheme. The proposed runtime system is implemented within PVFS2, a parallel file system. We conclude by a brief discussion of ongoing and future work on I/O.
Table of Contents

List of Figures vii
List of Tables xii
Acknowledgments xiii

Chapter 1
Introduction 1
1.1 Server Power Consumption 1
1.2 Disk power consumption 2
1.3 Contributions 4

Chapter 2
Power Management for Disk Subsystem 6
2.1 Introduction 6
2.2 Storage System Model 7
2.2.1 Implication of Abstraction in Multi-Layered Systems 9
2.3 Prior Work 9
2.4 Compiler-Based Disk Power Management 10
2.4.1 Disk Access Pattern Extraction 12
2.4.2 Proactive Disk Power Management 14
2.5 Experimental Evaluation 19
2.5.1 Setup 19
2.5.2 Benchmarks 21
2.5.3 Disk Power Management Schemes 21
2.5.4 Empirical Analysis 23
2.5.4.1 Base Results 23
List of Figures

1.1 The cost of power and cooling infrastructure [1]. . . . . . . . . . . . . 2
1.2 Contributions of each system component to the overall power con-
sumption in data centers. . . . . . . . . . . . . . . . . . . . . . . . . . . 3

2.1 Striping a file across multiple disks. . . . . . . . . . . . . . . . . . . 8
2.2 High-level view of our approach to proactive disk power manage-
ment. The dashed curve gives the boundary between the compiler-
related component and the simulation-related component of our
infrastructure. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.3 (a) Determining disk access pattern. (b) Striping an array over four
disks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.4 A data access pattern and the corresponding disk access pattern.
\(< d_i, t_j >\) means that disk \(d_i\) is (estimated to be) used for \(t_j\) cycles. 14
2.5 Comparison of the hardware based TPM and the proposed com-
piler directed TPM. In the hardware-based scheme, period \(T_w\) is
for detecting idleness and \(T_{su}\) is the spin-up latency. The compiler-
directed scheme can eliminate the impact of both these latencies. . 16
2.6 Compiler algorithm for inserting disk power management calls in a
given code fragment. . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.7 An example that illustrates proactive disk power management. . 18
2.8 Power mode transitions for the TPM disks. . . . . . . . . . . . . . . 20
2.9 CDF for idle times (periods). . . . . . . . . . . . . . . . . . . . . . . 24
2.10 Energy breakdown for the benchmarks. . . . . . . . . . . . . . . . . . . 24
2.11 Normalized energy consumptions. . . . . . . . . . . . . . . . . . . . 25
2.12 Normalized execution times. . . . . . . . . . . . . . . . . . . . . . . 25
2.13 Energy consumption with different stripe sizes. . . . . . . . . . . . . 27
2.14 Execution times with different stripe sizes. . . . . . . . . . . . . . . 27
2.15 Energy consumption with different stripe factor. . . . . . . . . . . . . 27
2.16 Execution times with different stripe factor. . . . . . . . . . . . . . . 27

3.1 Left: An example PDAG. Right: Two different scheduling. . . . . . 34
3.2 Example optimization schemes (low-power modes) ........................................ 34
3.3 An example application of the proposed scheduling approach .................. 35
3.4 Proposed scheduling algorithm with $P$ processors and $D$ disks ............. 37
3.5 An example that illustrates node merging and node partitioning .............. 38
3.6 Energy consumptions with different schemes .............................................. 46
3.7 Execution cycles with different schemes ..................................................... 46
3.8 CDF (cumulative distribution function) curves for disk idle times.
   An (x,y) point on a curve indicates that y% of the idle times has
   a duration of x (ms) or lower. As mentioned earlier, the minimum
   amount of idle time required to compensate the cost of spinning
   down the disk and up (under a TPM-based scheme) is called the
   threshold. Based on the numbers from IBM Ultrastar 36Z15, the
   threshold is 15.19 seconds. The results in this graph show that the
   idle disk times exhibited by these array-based applications are much
   shorter than the threshold value .................................................. 46
3.9 Impact of stripe size on energy consumption ............................................... 47
3.10 Impact of stripe factor on energy consumption .......................................... 47
3.11 Impact of starting disk on energy consumption ......................................... 49
3.12 Impact of processor count on energy consumption .................................... 49
3.13 Scalability of various schemes experimented ........................................... 50
3.14 Loop fission example. (a) Original code fragment. (b) Transformed
   code fragment. (c) Disk layout for the arrays ........................................ 52
3.15 Loop fissioning algorithm ........................................................................ 52
3.16 Example application of the proposed loop fission algorithm. (a)
   Original code fragment. (b) Transformed code fragment. (c) Disk
   allocation for array groups ................................................................ 52
3.17 Loop tiling algorithm ................................................................................. 54
3.18 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 .......... 55
3.19 Normalized energy consumptions ............................................................... 56
3.20 An example data access pattern scenario that involves four processors.
   In this scenario, three different loop nests access the same
   array ......................................................................................... 59
3.21 Two different parallelizations from the perspective of a given processor. (a)
   Loop-based parallelization. (b) Reuse-aware parallelization ...................... 60
3.22 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) .................................................................................... 63
3.23 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). ................................................................. 65
3.24 Example application of our disk layout-aware code parallelization. . 66
3.25 Normalized energy consumption results with the single processor execution. ................................................................. 70
3.26 Normalized energy consumption results with 4 processor execution. 70
3.27 CDF (cumulative distribution function) curves for disk idle times when TPM is used. An (x,y) point on a curve indicates that y% of disk idle times has a duration of x (ms) or lower. .................... 71
3.28 CDF curves for disk idle times when T-DRPM-s is used. ......... 71
3.29 Performance degradation results with the single processor execution. ................................................................. 73
3.30 Performance degradation results with 4 processor execution. .... 73
3.31 Normalized energy consumption results with different stripe factors (FFT; 4 processors). ................................................................. 73
3.32 Normalized energy consumption results with different start disks (FFT; 4 processors). ................................................................. 73
3.33 Normalized energy consumption results with different stripe sizes (FFT; 4 processors). ................................................................. 74
4.1 An example application of prefetching. .............................. 79
4.2 Comparison of I/O timings. (a) Original code fragment without prefetching. (b) Prefetching to high-speed disks. (c) Prefetching to low-speed disks. $T_d$ is the disk I/O time for a single block data. ... 81
4.3 Comparison of disk power states. (a) Original code fragment without prefetching. (b) Prefetching to high-speed disks. (c) Prefetching to low-speed disks. $T_d$ is the disk I/O time for a single block data. ................................................................. 81
4.4 Disk speed detection algorithm. ........................................ 85
4.5 Data layout detection algorithm. ..................................... 86
4.6 Code restructuring algorithm. ....................................... 87
4.7 The three steps of the proposed approach to energy-aware data prefetching. ................................................................. 87
4.8 Application of the proposed approach when data layouts and disk speeds are not adjustable. ................................................................. 88
4.9 An example application of our hoisting algorithm. ............... 89
4.10 Normalized energy consumptions. .................................. 93
4.11 Normalized execution times. ....................................... 93
4.12 Normalized energy consumptions with the different stripe sizes. . 95
5.1 Out-of-core matrix-matrix multiplication loop nest written using the MPI IO calls. Each data slice (tile) is of size $Nb \times Nb$ array elements, and all three matrices, A, B, and C, have $R \times R$ number of slices, which corresponds to the number of processors.

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

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

5.4 Disk power state diagram for the applications in Figure 5.3. (a) A1 and (b) A2. The disk is assumed to have four discrete speeds; 6K, 9K, 12K, and 15K.

5.5 The proposed runtime system that handles compiler-inserted disk power management calls.

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

5.7 CDF (Cumulative Distribution Function) for disk idle periods in our applications. An (x, y) point on a curve indicates that y% of the idle periods have a duration of x milliseconds (ms) or lower. The minimum amount of idle time required to compensate the cost of spinning down the disk and up is called the break-even time. Based on the numbers from the IBM36Z15 [2] disk data, the break-even time is 15.19 seconds. As most of idle periods are less then 1 second, this confirms that the spin-down disk is not a viable option for these applications. Instead, multi-speed disks are more promising.

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

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

5.11 Multiple application execution scenario with different set of applications.

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

5.13 Effect of caching to overall energy savings with the RT scheme.
List of Tables

2.1 Default simulation parameters. ........................................ 20
2.2 Benchmarks and their characteristics. ............................... 22
2.3 Disk speed misprediction rates. ....................................... 26

3.1 Default simulation parameters. ....................................... 42
3.2 Benchmarks and their characteristics. ............................... 42
3.3 Applications and their characteristics. ............................. 67

4.1 Major simulation parameters and their default values. .......... 91
4.2 Benchmarks and their characteristics. ............................... 93

5.1 Default System parameters. .......................................... 109
5.2 I/O applications used in our experiment. ........................... 110
5.3 Multiple application execution scenario. ........................... 114
Acknowledgments

First of all, I am deeply indebted to my thesis advisor, Dr. Mahmut Kandemir, for his guidance and support for the last five years at Penn State. Without his help, I cannot imagine how I could finish my dissertation. I also want to thank my thesis committee members, Dr. Padma Raghavan, Dr. Bhuvan Urgaonkar, and Dr. Long-Qing Chen, for their useful feedback to my dissertation. I thank Dr. Vijaykrishnan Narayanan for his encouragement and guidance.

I am very fortunate to spend two summers at an industry and a national laboratory as an intern. It was a pleasure and valuable experience working with Shiliang Hu and Youfeng Wu at Intel Programming Systems Laboratory. Both, Rajeev Thakur and Darius Buntinas at Argonne National Laboratory, were great mentors and gave me an opportunity to have an experience in real system software. Lastly and most importantly, I deeply thank my wife, Kathy, and my two children, Daniel and Ashley, who have always been an encouragement during my study. Without their timeless endurance and support, I could not finish my dissertation.

I would like to thank the National Science Foundation for funding my research over the years through various grants. I also would like to thank IITA (Institute for Information Technology Advancement) and MIC (Ministry of Information and Communication), Republic of Korea, for funding for my research through IT Scholarship Program.

Words cannot express my gratitude to my parents for their unconditional love, support, and encouragement. I also want to thank my brother, sister, and brother-in-law for their love and understanding.

Lastly and most importantly, I deeply thank my wife, Kathy, and my two children, Daniel and Ashley, who have always been an encouragement during my study. Without their timeless endurance and support, I could not finish my dissertation. I also want to thank my brother, sister, and brother-in-law for their love and support. Every other summer, she came to help my wife raise my two children.

I would like to thank the National Science Foundation for funding my research over the years through various grants. I also would like to thank IITA (Institute for Information Technology Advancement) and MIC (Ministry of Information and Communication), Republic of Korea, for funding for my research through IT Scholarship Program.
Chapter 1

Introduction

1.1 Server Power Consumption

Historically, the cost of energy and the cost of the data center power and cooling infrastructure have not been considered in TCO (Total Cost of Ownership) models [3]. Therefore, industry has been focused on reducing the cost of IT equipment in the data center by employing faster processors, memory, and disk drives. This leads to increase in server power consumption. For instance, the power consumption by a 1U server has been increased from 100–200 Watt to 300–400 Watt between 2000 and 2006 [3]. In addition, data centers are generating larger head loads due to the increase in power consumption. As indicated by the ASHRAE power trends curve [4], power density has been increasing by approximately 15% annually as systems become more dense.

Therefore, power consumption of large-scale parallel systems has recently been a popular research topic. There are at least three reasons for that. First, these systems are power hungry as documented by the prior work, which indicates that they can consume several mega-watts of power [5, 6, 7]. As shown in Figure 1.1, the cost of power and cooling has been estimated to increase significantly for the last decade and is expect to grow to the level comparable to the new server spending. There are several other documents from industry that report similar trends in power consumption of data center [3, 8, 9].

Second, the cooling systems required for these systems can be extremely costly, which in turn contributes to expensive electrical bills [10]. A recent EPA (Envi-
ronmental Protection Agency) report to U.S. Congress estimated that U.S. servers and data centers used about 61 billion kilowatt-hours of electricity in 2006 at a cost of about 4.5 billion dollars, which corresponds to 1.5% of the total electricity used that year [11]. A more recent report also estimated that the overall electricity used by servers doubled between 2000 and 2005. This doubled growth rate is mainly due to an increase in the number of servers installed in data centers and the increasing demands of cooling equipment [12].

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. This might be more important to protect our environment because major sources of power generation (coal, nuclear, natural gas, etc) are anti-environment. In addition, air pollution from diesel generators activated when the electrical grid is unstable or unavailable may cause several health-related problems [13].

### 1.2 Disk power consumption

There are several components in large-scale computing systems that contribute overall power consumption, and storage subsystem is known to be one of major energy consumers [14, 15, 16]. For example, as shown in Figure 1.2, disk subsystem can contribute up to 30% of total energy consumed by large-scale server systems [15]. More recent report by IDC also indicates that the costs to businesses of
powering cooling their external storage reached 1 billion dollars in 2007 [14].

The main reason for this is the explosive increase in disk drive in server systems. According to a recent IDC report, the storage requirements for large-scale server systems are growing at about 55% a year, while hard disk drive capacities are growing at 30 to 35% a year [14]. This indicates that we need to put more disk drives into the large-scale server systems. Between 2008 and 2012, the storage industry will ship almost eight times the amount of external storage that is currently sold, according to IDC [14].

In petascale scientific computing environments, the computational problems are becoming data-intensive [17]. This means that next generation computational systems will generate petascale volumes of datasets. In order to balance between a petascale processor and I/O subsystem, the data (both experimental and simulated) needs to be stored in a million disks to support the I/O needs. Once scientific data is captured and stored, huge computational powers are needed to analyze and visualize the data. Both analyzing and visualizing petabytes of data require huge amount of storage capacity and I/O bandwidth. That generates even more I/O. Apart from I/O performance needs, these data should be stored reliably, which is usually achieved by employing storage arrays.

Such steep growth rate, however, brings the problems of high power consump-

Figure 1.2. Contributions of each system component to the overall power consumption in data centers.
tion. The increase in power consumption in storage systems is mainly attributed to the largest 15,000 RPM drives. In general, a 3.5-inch 15,000 RPM drive consumes more power than a 2.5-inch 15,000 RPM drive because there are more spindles and bigger (heavier) platters. Therefore, the number of spinning disks in external enterprise storage continues to increase every year, and there is a cost beyond just acquiring these disks - the cost to power and cool them.

1.3 Contributions

Focusing on scientific applications that manipulate large, disk-resident data, this dissertation proposes a series of software-based disk power management schemes. More specifically, it explores how to make the conventional disk power management schemes more effective through the aid of compiler and runtime system support.

To reduce the energy consumption of the disk subsystems, this dissertation first proposes a compiler-based proactive disk power management scheme for sequential and parallelized applications. In the proposed approach, it enlists 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 prior research. First, since (in the context of array-based scientific 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 systems that employ disks of variable speeds), and this in turn helps reduce 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 patterns. Therefore, this dissertation focuses on array-based scientific applications where source codes are accessible to and analyzable by an optimizing compiler.

Increasing disk idle periods is crucial regardless of whatever the underlying hardware disk power management scheme employed because such enlarged idle
period is more exploitable than shorter ones. To achieve this, we propose several compiler-based code transformation approaches to improve the effectiveness of the 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.

Although full-blown multi-speed disks are not available in the market, they bring more opportunities when they are used with a prefetching technique, which is known to be very effective in hiding I/O latency [18]. To exploit this, we propose an energy-aware prefetching scheme that determines the most preferable prefetch distances for each reference to a disk-resident array, disk speeds (RPM levels) for all the disks in the storage system, and corresponding disk layouts in a unified fashion.

When multiple applications use the same set of disks concurrently, a runtime system-based approach may be more suitable for reducing disk power consumption. Motivated by this observation, this dissertation proposes a runtime system centric disk power reduction scheme in which the runtime system receives, as hints, the preferred disk speeds from individual applications and, considering all hints, decides the best rotational speed for each disk in the system such that overall power consumption is reduced without affecting performance.

The remainder of this dissertation is organized as follows. The compiler-based proactive schemes are discussed in Chapter 2. Chapter 3 introduces several code transformations/restructuring schemes that improve the effectiveness of compiler-based disk power management schemes. Chapter 4 presents the energy-aware prefetching for multi-speed disks. In Chapter 5, we present a runtime system based disk power reduction scheme. Finally, Chapter 6 concludes with the summary of our major contributions, and a brief discussion of the ongoing and future research agenda.
Chapter 2

Power Management for Disk Subsystem

2.1 Introduction

One way of reducing excessive power consumption by disk subsystem is to adopt architectural mechanisms such as spinning down idle disks [19, 20, 21] or rotating disks with reduced speed [22, 10] when some amount of latency can be tolerated. A review of the prior work on disk power management is given in Section 2.3. While such techniques have been shown to be effective in certain cases, they have a common drawback: they operate in a reactive manner, that is, they control disk behavior based on observed disk activity (e.g., idle and active periods). In practice, this can bring two problems. First, they may fail to select the most appropriate disk power management scheme since their disk idleness estimation can be inaccurate. For example, if disk idleness is underestimated, these schemes behave conservatively in selecting the low-power mode to be employed. Consequently, they may not be able to use the most aggressive low-power mode. Second, they can incur performance penalties if they cannot determine accurately when an idle disk is going to be needed in the future. This is one of the most pressing problems facing parallel systems where disk requests coming from individual processors can interleave in time, and eventually make disk idle time (and active time) prediction very difficult.
Motivated by these observations, this chapter proposes and experimentally evaluates a \textit{compiler-directed} disk power management scheme targeting array-based scientific parallel applications executing on environments with parallel disks. An optimizing compiler is in a very good position for the application domain and execution platform stated above. This is because the compiler can analyze data access pattern of a scientific application based on a high level representation of the program [23, 24], which enables us to capture how the disk-resident data are accessed and shared by parallel processors. As for determining disk idle and active periods, extracting data access pattern alone may not be sufficient, and one actually needs the \textit{disk access pattern}. We propose to obtain this pattern by \textit{exposing} the layout information of disk-resident data to the compiler. In other words, the proposed compiler support obtains disk access pattern by using data access pattern and disk layout information for array data. Section 2.4.1 explains the proposed disk access pattern extraction process in detail.

After extracting the disk access pattern, this information can be used for implementing a \textit{proactive} disk power management strategy. What we mean by this is to let the compiler decide the times at which disks are switched to a low-power operating mode (e.g., spinning down a disk or operating it under reduced speed) and restored to the active status. As will be demonstrated in this chapter, this proactive scheme can bring significant additional power benefits over the state-of-the-art hardware-based reactive power management strategies.

### 2.2 Storage System Model

The storage and file system architecture considered in this dissertation is depicted in Figure 2.1. Most parallel file systems available today either commercially or for research purposes provide high-performance I/O by \textit{striping} (i.e., dividing a file into blocks called stripes and distributing these blocks in a round-robin fashion across multiple disks), as illustrated in Figure 2.1, to support parallel disk accesses for I/O-intensive applications that manipulate disk-resident, multidimensional arrays. In such a system, the disk access pattern of an application is strongly influenced by the striping parameters. In Figure 2.1, the data file, foo, is striped across three I/O nodes, starting from node 1. Within each I/O node, a stripe assigned to that
I/O node can be further striped at the RAID level (depending on the RAID level [25] adopted) for performance and reliability purposes. Therefore, in many parallel file systems, a given file is typically striped at two different layers, i.e., hardware layer through RAID and software layer through file striping. While the hardware level striping is hidden from the compiler and the operating system, the file system level striping can be exposed to the software. Furthermore, the file level striping can even be controlled through the APIs provided by the file system used. For example, PVFS2 [26] provides an MPI hint mechanism that allows application programmers to change the default stripe distribution across the available I/O nodes. Other parallel file systems such as PFS [27] and GPFS [28] also provide similar facilities.

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

In this dissertation, 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 consider these options in this dissertation. We also 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.

### 2.2.1 Implication of Abstraction in Multi-Layered Systems

Current approaches to disk power managements, including ours, are based on operating systems or runtime systems with full knowledge of underlying hardware, e.g., file to disk mapping. The abstract nature of multi-layered environments, however, makes such approaches inappropriate. For example, over the past few years, virtualization technology has received considerable attention as a solution to increasing resource utilization and maintainability in designing modern storage systems [29, 30]. The virtualized environments are typically achieved by hiding the actual complexity of the physical disk subsystem. In such an environment, direct control of underlying disk subsystem is infeasible. Similarly, in the scientific application domain, many applications are written using a set of software libraries such as NetCDF [31, 32] and HDF [33, 34] that provide machine-independent data formats for creating, accessing, and sharing array-oriented scientific data. These additional abstraction in the software stack also hinders gaining full control over hardware devices and their modes of operations. Therefore, in a multi-layered systems, power optimization in the highest-level should be distributed across whole system. In a hypervisor-based virtual systems, one can adapt the distributed OS power management that models the OS as a set of components, each of which is responsible for managing underlying hardware devices [35].

### 2.3 Prior Work

Disk power management has been extensively studied in the context of laptop and desktop disks [19, 20, 21]. 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 dissertation), 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. A recent study considered program counter based techniques for predicting upcoming disk accesses [36]. While TPM is an effective approach in the domain of laptop/desktop systems, recent studies [22, 10] 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) [22]. 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 disks (e.g., those from [37], [38], and [39]) 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 [10]. When there is no confusion, in the rest of this chapter, we use the term “low-power mode” 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).

### 2.4 Compiler-Based Disk Power Management

To address excessive power consumption in disk subsystem, prior studies proposed several techniques, such as TPM and DRPM. 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 behavior observed during execution. While such
techniques are certainly useful in some cases, one can conceivably achieve better energy savings by adopting a proactive scheme.

Our overall approach to proactive disk power management is depicted in Figure 2.2. A unique characteristic of our approach is that it exposes the disk parameters and the disk layout of array data to the compiler. The goal of doing so is to allow the compiler to determine (estimate) the disk active and idle times. The parameters used by the compiler in this approach include the number and ids of the disks over which each array is striped (i.e., the stripe factor), the stripe size used, and the id of the disk that contains the first stripe of data. Using this information and the data access pattern extracted by analyzing the application source code, the compiler determines what we refer in this chapter as the disk access pattern (DAP). Basically, a DAP indicates how the disks in the I/O subsystem are accessed and reveals an information that is vital for disk power management: disk inter-access times, i.e., the gap between the two successive accesses to a given disk. This information can be used in two different ways to do proactive power management: (1) placing a disk into a suitable low-power mode after its current access is complete, and (2) pre-activating a disk to eliminate the potential performance penalty due to power management. Then, based on this information, the application code is modified to insert explicit power management calls. The nature of these calls depends on the underlying method used (e.g., TPM versus DRPM), and will be explained shortly. This compiler-transformed code can execute on a system that uses TPM or DRPM disks to do proactive power management. In
the rest of this section, we explain the two important components of our compiler-driven approach: the extraction of disk access pattern and the insertion of explicit power management calls into the code.

2.4.1 Disk Access Pattern Extraction

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

One of the requirements for being able to use a compiler in reducing disk power consumption is to capture how parallel disks are accessed at a high level (i.e., source code level). We use the term disk access pattern in this dissertation to refer to the high-level information on the order in which parallel disks are accessed by a given application code. This order is important since it determines, for each disk in the system, active and idle periods, which is the primary information used for power management as will be explained in Section 2.4.2. Disk access patterns can be extracted at the loop iteration, loop nest, procedure, or even larger granularity. To obtain this information, the compiler needs data access pattern of the application code being optimized and disk layout information for array data (see Figure 2.3(a)). The first of these can be obtained by analyzing the application source code. Since such an analysis is performed by many optimizing compilers for different purposes (e.g., optimizing loop-level parallelism or cache locality), we do not discuss its details in this dissertation. As for the second parameter needed, we propose to expose the disk layout information to the compiler. In this way, the compiler will
be aware of how array data is striped across the parallel disks, and can optimize (i.e., restructure) the code accordingly.

We next discuss what type of disk layout abstraction is needed by the compiler in the proposed approach. File striping is a technique that divides a large data into small portions and stores these portions on separate disks in a round-robin fashion (as depicted in Figure 2.3(b)). This permits multiple processes to access different portions of the data concurrently without much disk contention. While striping can be performed manually, many file systems today provide automatic support for it, as will be explained below. In this dissertation, we represent disk layout of an array using a triplet of the form:

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

The first component in this triplet indicates the disk from which the array is started to get striped. The second component gives the number of disks used to stripe the data, and the third component gives the stripe (unit) size. Note that the several current file systems and I/O libraries for high-performance computing provide APIs to convey them the disk layout information when the file is created. For example, in PVFS [40], one can change the default striping parameters by setting `base` (the first I/O node to be used), `pcount` (stripe factor), and `ssize` (stripe size) fields of the `pvfs_filestat` structure. Then, the striping information defined by the user via this `pvfs_filestat` structure is passed to the `pvfs_open()` call's parameter. When creating a file within the application, this layout information can be made available to the compiler as well, and, as explained above, the compiler uses this information in conjunction with the data access pattern it extracts to determine the disk access pattern. On the other hand, if the file is already created on the disk system, the layout information can be passed to the compiler as a command line parameter.

The important point to note here is that we assume each data array manipulated by the application is stored in a separate file in the I/O system\(^1\). Since each file can have a different triplet of the kind shown above, each array can have a different disk layout than the others. While determining power-efficient disk

\(^1\)The proposed approach can be modified to handle other scenarios as well, e.g., multiple arrays per file, or multiple files per array.
Data Access Pattern

2, 3, 4, 5, 0, 1, 2, 3, 15, 12

Disk Layout

0, 1, 2, 3 ⇔ d_0
4, 5, 6, 7 ⇔ d_1
8, 9, 10, 11 ⇔ d_2
12, 13, 14, 15 ⇔ d_3

Disk Access Pattern

<d_i, t_j>, <d_i, t_j>, <d_i, t_j>, <d_i, t_j>

Figure 2.4. A data access pattern and the corresponding disk access pattern. <d_i, t_j> means that disk d_i is (estimated to be) used for t_j cycles.

layouts itself is an interesting research topic that we want to tackle in the future, in this dissertation, we concentrate on code restructuring for low power. As a consequence, we assume that the disk layout information is given to the compiler, which subsequently uses it for determining disk access patterns.

Figure 2.4 shows a sample data access pattern and the corresponding disk access pattern. This disk access pattern is obtained under the disk layout shown in the same figure. In this layout, for illustrative purposes, the twelve elements of an array are distributed (striped) across four disks (d_0 through d_3). In the disk access pattern, a <d_i, t_j> means that disk d_i is used for t_j cycles. t_j is estimated by the compiler. It is to be noted that the compiler can represent a disk access pattern using different representations and with different granularity. Since a given disk access pattern captures idle and active periods for each disk and their durations, it can be used for proactive power management (Section 2.4.2) or to restructure code to increase idle periods (Chapter 3).

2.4.2 Proactive Disk Power Management

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

spin_down(d_i),

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

\[
Q_{su} = \left\lceil \frac{T_{su}}{s + T_m} \right\rceil,
\]

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

\[
\text{spin\_up}(d_i),
\]

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

\[
\text{set\_speed}(\text{rpm\_level}_j, d_i),
\]

where \(d_i\) is the disk id, and \(\text{rpm\_level}_j\) is the \(j^{th}\) RPM level (i.e., disk speed) available. When executed, this call brings the disk in question to the speed specified. The selection of the appropriate disk speed is made as follows. Since the transition time from one RPM step (level) to another is proportional to the difference between the two RPM steps involved [22], we need to consider the detected idle
Figure 2.5. Comparison of the hardware based TPM and the proposed compiler-directed TPM. In the hardware-based scheme, period $T_w$ is for detecting idleness and $T_{su}$ is the spin-up latency. The compiler-directed scheme can eliminate the impact of both these latencies.

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

Figure 2.5 illustrates the difference between the hardware-based TPM and the compiler-directed TPM. Compared to the hardware-based TPM, the proposed approach has two advantages. First, the compiler-directed TPM can put idle disks
procedure loopTransformation() {
    buildLTG();
    transform();
}

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

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

procedure transform() {
    for each node \( L_i \) in the LTG
        if\((\text{execTime}(L_i) > Q)\)
            for each disk \( d_x \)
                if\((D_i[x] = 0)\)
                    insert before the entry of loop nest \( L_i \): "spin\_down\((d_x)\)";
                if\((\text{exists } L_i \text{ and } L_j \text{ such that } d[j] \neq d[i])\)
                    split \( L_i \) into two consecutive loop nests: \( L'_i \) and \( L''_i \)
                    such that \( \text{execTime}(L''_i) = Q_{su} \);
                    for each disk \( d_x \) such that \( D_i[x] = 0 \)
                        for each loop nest \( L_j \) such that \( L_i \xrightarrow{C_{i,j}} L_j \)
                            if\((D_j[x] = 1)\)
                                insert before the entry of \( L''_i \): "if\((C_{i,j}) \text{ spin\_up\((d_x)\)}\)";
}

Figure 2.6. Compiler algorithm for inserting disk power management calls in a given code fragment.

...
loop nest whose execution time is longer than a given threshold $Q$ is recursively broken down into smaller loop nests until no loop nest contains any internal loop, or the execution time of the loop is shorter than $Q$. Each edge (from $L_i$ to $L_j$) in LTG has a tag $C_{i,j}$, indicating the condition under which the flow of execution transitions from loop nest $L_i$ to $L_j$. Figure 2.7(b) shows an LTG for the code fragment in Figure 2.7(a). In the second step, the proposed algorithm inserts code to the program to spin up/down the disks. Specifically, for each node $L_i$ in LTG, the algorithm inserts, before the entry of $L_i$, the spin-down calls for the disks that are not accessed in $L_i$. Further, if node $L_i$ has a successor $L_j$ that accesses a disk that has been spun down in $L_i$, it splits $L_i$ into two consecutive loop nests, $L_i'$ and $L_i''$, such that the execution time of $L_i'$ is equal to $Q_{su}$, the time required to spin up a disk. Before $L_i''$, the algorithm inserts the spin-up calls for the disks that will be used in $L_j$. By doing this transformation, we can hide the performance overhead due to disk spin up. That is, as explained earlier, this pre-activation eliminates
potential performance penalty. Figure 2.7(a) shows an example code fragment, and Figure 2.7(b) gives the corresponding LTG. Figure 2.7(c) is the transformed code fragment after applying the proposed algorithm.

2.5 Experimental Evaluation

2.5.1 Setup

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 such as stripe unit size, striping factor (the number of disks), and starting iodevice (disk). 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. The default stripe size is 64KB but the range can be varied from 16KB to 256KB. In our simulator, the striping information is provided in an external file along with other parameters. The default simulation parameters are given in Table 2.1.

To evaluate our approach and the prior proposals to disk power management, we developed a simulation platform using DiskSim [41]. The trace file generated from our trace generator gives time stamps of each I/O request. While processing the given input traces, our simulator keeps track of the state of each disk (i.e., when a particular disk is used and how long it is used) throughout the entire simulation time, and then we calculate power values from these determined states of each disk. The simulator also generates statistical data for performance. Both performance and energy statistics were calculated based on the figures extracted from the data sheet of the IBM Ultrastar 36Z15 [2], and are given in Table 2.1. The power mode transition diagram for IBM Ultrastar 36Z15 is given in Figure 2.8. Note that the mode transitions shown in Figure 2.8 are only applicable to TPM. Because we are primarily interested in the performance and energy consumption
Table 2.1. Default simulation parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk Model</td>
<td>IBM Ultrastar 36Z15</td>
</tr>
<tr>
<td>Interface</td>
<td>SCSI</td>
</tr>
<tr>
<td>Storage Capacity</td>
<td>18.4 GB</td>
</tr>
<tr>
<td>Number of Platters</td>
<td>4</td>
</tr>
<tr>
<td>RPM</td>
<td>15,000</td>
</tr>
<tr>
<td>Disk controller cache</td>
<td>4 MB</td>
</tr>
<tr>
<td>Average seek time</td>
<td>3.4 msec</td>
</tr>
<tr>
<td>Average rotational latency</td>
<td>2 msec</td>
</tr>
<tr>
<td>Internal transfer rate</td>
<td>55 MB/sec</td>
</tr>
<tr>
<td><strong>Parameters specific to TPM</strong></td>
<td></td>
</tr>
<tr>
<td>Power (active)</td>
<td>13.5 W</td>
</tr>
<tr>
<td>Power (idle)</td>
<td>10.2 W</td>
</tr>
<tr>
<td>Power (standby)</td>
<td>2.5 W</td>
</tr>
<tr>
<td>Energy (spin down: idle → standby)</td>
<td>13 J</td>
</tr>
<tr>
<td>Time (spin down: idle → standby)</td>
<td>1.5 sec</td>
</tr>
<tr>
<td>Energy (spin up: standby → active)</td>
<td>135 J</td>
</tr>
<tr>
<td>Time (spin up: standby → active)</td>
<td>10.9 sec</td>
</tr>
<tr>
<td><strong>Parameters specific to DRPM</strong></td>
<td></td>
</tr>
<tr>
<td>Maximum RPM level</td>
<td>15,000 RPM</td>
</tr>
<tr>
<td>Minimum RPM level</td>
<td>3,000 RPM</td>
</tr>
<tr>
<td>RPM Step-Size</td>
<td>1,200 RPM</td>
</tr>
<tr>
<td><strong>Stripe Information</strong></td>
<td></td>
</tr>
<tr>
<td>Stripe unit (stripe size)</td>
<td>64 KB</td>
</tr>
<tr>
<td>Stripe factor (number of disks)</td>
<td>8</td>
</tr>
<tr>
<td>Starting iodevice (starting disk)</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2.8. Power mode transitions for the TPM disks.

of the disk subsystem, we assume that other performance enhancement techniques like I/O prefetching [18] are not employed.

For DRPM, we obtained statistics using the model described in [22]. The simulation parameters specific to DRPM are also given in Table 2.1. Since the lowest RPM step (level) used is 3,000 and a single RPM step is 1,200, we have a total of eleven RPM steps. Besides the parameters and values given in Table 2.1, we obtained the RPM transition time and energy consumption at each RPM level as well. Based on our calculations, the RPM transition time from the peak RPM level (15,000) to the lowest RPM level (3,000) was approximately 8.72 seconds. 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 [22]. We used the same heuristic algorithm in implementing DRPM for both upper and lower tolerance. In the rest of this chapter, 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.

### 2.5.2 Benchmarks

Table 2.2 gives the set of array-based benchmark codes used in this study. These benchmarks were selected randomly from the SPECfp2000 benchmark suite [42]. We made the data manipulated by these benchmarks disk-resident. As a result, each array reference causes a disk access unless the data requested is captured in the buffer cache. Note that, we hand-optimized all benchmarks (which is common for large I/O-intensive applications) in such a way that all benchmarks make use of the buffer cache fully. Therefore, the energy savings that will be presented later in this section are achieved from our schemes only. Also, to complete our simulations within a reasonable time frame, we focused only on time-consuming loop nests of 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 of Table 2.2 gives a brief description of each benchmark. The third column of this table gives the total dataset size manipulated by the selected nests. 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 chapter are with respect to the values listed in these last two columns of Table 2.2.

### 2.5.3 Disk Power Management Schemes

To compare different approaches to disk power management, we implemented and performed experiments with different schemes:
<table>
<thead>
<tr>
<th>Name</th>
<th>Brief Description</th>
<th>Data Size (GB)</th>
<th>Base Energy (J)</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>wupwise</td>
<td>Physics/Quantum Chromodynamics</td>
<td>94.3</td>
<td>37436.1</td>
<td>447.0</td>
</tr>
<tr>
<td>swim</td>
<td>Shallow Water Modeling</td>
<td>93.8</td>
<td>42223.6</td>
<td>504.3</td>
</tr>
<tr>
<td>mgrid</td>
<td>Multi-grid Solver: 3D Potential Field</td>
<td>48.2</td>
<td>20130.8</td>
<td>204.5</td>
</tr>
<tr>
<td>applu</td>
<td>Parabolic / Elliptic Partial Differential Equations</td>
<td>53.4</td>
<td>20647.4</td>
<td>246.5</td>
</tr>
<tr>
<td>mesa</td>
<td>3-D Graphics Library</td>
<td>70.3</td>
<td>32695.5</td>
<td>390.7</td>
</tr>
<tr>
<td>galgel</td>
<td>Computational Fluid Dynamics</td>
<td>78.1</td>
<td>29937.7</td>
<td>357.4</td>
</tr>
</tbody>
</table>

**Table 2.2.** Benchmarks and their characteristics.

- **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 2.2).

- **TPM:** This is the traditional disk power management strategy used in studies such as [19] and [20]. In this version, a disk is spun down after fixed threshold time 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 times.

- **Ideal TPM (ITPM):** This is the ideal version of the previous strategy. In this scheme, we assume the existence of an oracle predictor for detecting idle periods. Consequently, the spin-up and spin-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 [22]. Considering the predicted length of the idleness, it sets the rotation speed of the disk to an appropriate level to save power. Therefore, it is effective in saving power even if the idle periods are short. Note that, the RPM level used is selected based on the estimated idleness, and we may incur a performance penalty.
• **Ideal DRPM (IDRPM):** This is the ideal version of the previous 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 scheme 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 compiler-based schemes come close to the optimal. Also, in our experimental evaluation, TPM operates in a way that it uses the threshold value (for each application) that generates the best energy saving. Similarly, for the DRPM experiments, we used a window-size of 100, which gives the best energy savings for all applications tested during experiments. All necessary code modifications are automated using the SUIF infrastructure [43].

### 2.5.4 Empirical Analysis

#### 2.5.4.1 Base Results

Before discussing the behavior of different approaches to disk power management, let us first present some disk-related statistics for the array-based applications used in this study. The graph in Figure 2.9 gives the CDF (cumulative distribution function) for disk idle times. Specifically, an (x,y) point on a curve in this graph indicates that y% of the idle times has a duration of x (ms) or lower. As mentioned earlier, the minimum amount of idle time required to compensate the cost of spinning down and the disk and up (under a TPM-based scheme) is called
the break-even threshold [10]. As an example, based on the numbers from IBM Ultrastar 36Z15, the break-even threshold is 15.19 seconds. Now, the results shown in Figure 2.9 clearly show that the idle disk times exhibited by these array-based applications are much smaller in length. In fact, based on these results, it does not seem possible to take advantage of any TPM-based mechanism (including the compiler-driven one) to save disk power. These results motivate a DRPM-based approach to disk power management for array-based scientific codes. While the individual idle times may not be very large, they are numerous (as observed by [22] as well for synthetic traces); consequently, as shown in the bar-chart in Figure 2.10, they constitute a large fraction of the overall disk energy consumption. The results given in this bar-chart are for the case when no power optimization is used (i.e., the base version). We see that the energy consumed in the idle cycles is more than 9 times of the energy consumed in the active cycles. These results also favor DRPM over TPM.

The graph in Figure 2.11 gives the energy consumption of our benchmarks under the different schemes described earlier. One can make several observations from these results. First, as expected from the profile in Figure 2.9, 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 ideal DRPM (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 2.12 gives the normalized execution times (with respect to the base version) for the different schemes evaluated. The reason that the TPM-based schemes do not incur any performance penalty is because 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 2.11 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
Table 2.3. Disk speed misprediction rates.

<table>
<thead>
<tr>
<th></th>
<th>wupwise</th>
<th>swim</th>
<th>mgrid</th>
<th>applu</th>
<th>mesa</th>
<th>galgel</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMDRPM</td>
<td>6.78 %</td>
<td>5.14 %</td>
<td>13.02 %</td>
<td>18.97 %</td>
<td>27.35 %</td>
<td>15.9%</td>
</tr>
</tbody>
</table>

and eliminates the performance penalty. To better explain why our compiler-driven approach (CMDRPM) comes close to the ideal DRPM (IDRPM), we give in Table 2.3 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 DRPM. We see that the percentage mispredictions are not very large, which explains the success of the compiler-driven scheme.

2.5.4.2 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 part of our experimental evaluation, we vary the values of these parameters to see how our savings are effected. For illustrative purposes, we chose one benchmark, swim, and performed all sensitivity analysis with this. Figures 2.13 and 2.14 give the normalized energy consumptions and execution times, respectively, with the different stripe sizes. The values of all other simulation parameters are as given in Table 2.1. 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, the service duration for a particular disk increases, 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 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 2.15 and 2.16 give the normalized energy and execution time results, respectively, with the different stripe factors. As before, all other parameters are set to their default values given in Table 2.1. 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 ideal DRPM.
2.6 Summary

Excessive power consumption is becoming a major barrier to extracting the maximum performance from high-end parallel systems. Therefore, techniques oriented towards reducing power consumption of such systems are expected to become increasingly important in the future. Since disk subsystems of parallel architectures are known to consume a large fraction of the overall power budget, they are an important optimization target. Unfortunately, most of the prior work on disk power management focused exclusively on hardware-based approaches that operate with past history information collected during execution. In contrast, this chapter proposes a compiler-driven approach to disk power management for data-intensive scientific applications. The compiler in the proposed approach derives data access pattern and, by combining this information with the disk layout of array data, it obtains the disk access pattern. This extracted information is then used to make proactive disk power management decisions. The experimental analysis with several applications that operate with disk-resident data sets are very promising and show that the proposed compiler-driven approach performs much better than existing hardware-based techniques.
Chapter 3

Code Transformations and Restructuring for Reducing Disk Power Consumption

3.1 Introduction

The discussion 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 [23, 24] that loop transformations are very effective in optimizing data locality (mainly cache behavior) and iteration-level parallelism, there are not many studies that uses code restructuring for minimizing disk power consumption. This chapter presents three code restructuring approaches for minimizing disk power consumption by large-scale scientific applications: code scheduling for increasing disk idle times, energy-aware loop transformations, and increasing disk reuse for energy savings.
3.2 Related Work

There are three categories of studies that considered the way of increasing disk idle periods in a different layer of the systems. The first category investigates schemes to increase disk idle periods by delaying or prefetching I/O requests. Papathanasiou and Scott proposed energy-aware caching and prefetching scheme [44] for a single disk with laptop applications, which typically shows longer idle periods. Zhu et al, on the other hand, proposed two schemes, called PA-LRU [45] and PB-LRU [46], focusing on multiple disks with data center workloads. Heath et al [47] considered a scheme that transforms application code for increasing idle periods for a single disk with laptop workloads.

The next category considered energy-efficient disk arrays to trade off availability against disk energy by spinning down over-provisioned disk drives depending on the workload variation. These techniques include EERAID [48] and PARAID [49]. Zhu et al [50] recently proposed a scheme, called Hibernator, that combines disk block reorganization with dynamic disk speed setting and multi-tier data layout. Their idea is to maintain several groups of disks that are spinning at different speeds and adjust disk layouts and disk speeds dynamically depending on the response time variation. More recently, Cai et al [51, 52] discussed a power management scheme that considers reducing memory and disk energy simultaneously under certain performance constraints.

The last category investigates file-level data reorganization schemes that cluster disk accesses onto a subset of disks. MAID (Massive Arrays of Idle Disks) [53] uses a small number of cache disks in maintaining recently accessed blocks, thereby reducing the number of spin-ups for the remaining disks. PDC (Popular Data Concentration) [54] migrates the most frequently accessed files to a subset of the disks in the array. Both MAID and PDC perform well with workloads that are heavily skewed toward a small set of files. Because MAID and PDC move frequently accessed files to a small number of disks, they can incur performance degradation due to heavily loaded disk I/O load.
3.3 Code Restructuring for Reducing Disk Energy Consumption

In this section, we present a strategy that restructures a given procedure for increasing the benefits that could be obtained from the proactive scheme discussed in Chapter 2. This code restructuring approach operates on a graph representation called the Inter-Processor Disk Access Graph (or IDAG for short). An IDAG is composed of a number of Processor Disk Access Graphs (PDAGs). Each node in an IDAG represents a set of loop iterations (as will be explained shortly), and the directed edges between nodes capture data dependences.

We assume that the set of loop iterations that will be executed by each processor has already been determined prior to approach. For this purpose, either user-assisted (e.g., [55]) or compiler-directed (e.g., [56]) code parallelization methods can be employed. The selection of the method to be used for assigning loop iterations to parallel processors in the system is orthogonal to the focus of this dissertation. Let $I_p$ represent the set of iterations assigned to processor $p$ (as a result of loop parallelization), where $0 \leq p \leq P - 1$. We note that, for any legal parallelization scheme, we have:

$$\bigcup_{p=0}^{P-1} I_p = I_{total},$$

where $I_{total}$ is the set of total iterations in the procedure (including all the loop nests).

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

---

1The proposed approach is conservative in the sense that if $I$ may access disk $d$ (depending on conditional execution flow at runtime), it conservatively set the corresponding bit to 1.
terms, we have:

$$\mathcal{I}_{p,T} = \{ I \mid I \in \mathcal{I}_p \land \text{tag}(I) = T \},$$

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

Note that, from the disk power management perspective, it is beneficial to execute iterations in $$\mathcal{I}_{p,T}$$ one after another. This is because all the iterations in this set access the same set of disks, and the remaining disks can be placed into a low-power mode during these accesses to save power. However, it is also important to determine a good execution order for different $$\mathcal{I}_{p,T}$$s. In Sections 3.3.1 and 3.3.2, we present scheduling schemes, where the problem is considered from single processor’s perspective and multi-processors’ perspective, respectively. What we mean by “scheduling” in this context is an order in which the nodes in an IDAG (or PDAG when considering from the perspective of a single processor) are executed. In Sections 3.3.1 and 3.3.2, we explain the proposed approach, assuming that PDAGs (or IDAG) in question are cycle-free. Later in Section 3.3.3, we discuss code transformations to eliminate cycles in IDAG/PDAGs. After these code restructurings, the resulting code is further modified by inserting the proactive disk power management calls as has been discussed in Chapter 2 (see Section 2.4.2).

### 3.3.1 Single Processor Perspective

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

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

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

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

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

If I_{p,T} and I_{p,T'} are the two nodes (in PDAG_p) that are successively visited where T and T' are the respective tags, the variation in disk activation and disk idleness patterns in going from I_{p,T} to I_{p,T'} is a function of the Hamming distance between T and T'.

For instance, in an I/O system with eight disks, if we schedule I_{p,01101010} and I_{p,01100101} one after another, the first four disks preserve their states (during this transition), whereas the remaining four disks change their states. Minimizing the Hamming distance between the tags of classes that are visited successively is useful in reducing the disks energy consumption. In other words, for a given disk in the I/O system, in going from one class (node) to another, it is better to keep the states of the disks (active or idle) similar as much as possible. This is because if the first state is 0 and the second is also 0, the disk in question will have a long idle period (which is good from an energy consumption viewpoint); and similarly, if both the states in question are 1, this means that the active periods are clustered together; so, we will also have clustered idle periods for the disk (later when we visit the remaining classes). Based on this observation, from the viewpoint of a single processor (p), the problem of reducing disk energy consumption becomes one of scheduling a group of nodes taking accounts of some constraints (inter-class dependences) to minimize (optimize) some objective function (minimizing the Hamming distance between the number of successively visited classes).
To demonstrate how such a scheduling can be beneficial, let us consider the PDAG shown on the left side of Figure 3.1. Each node is annotated using its tag (assuming an I/O system with 4 disks). The column titled “Random” on the right side of Figure 3.1 gives a legal schedule, wherein the next node to be scheduled is selected randomly (by observing the dependences though). Assume that each node takes the same amount of time. Assume further that we have three power optimization schemes that operate as follows (see Figure 3.2). The first scheme ($S_1$) is applicable when we have, for a disk, two consecutive “0”s in the schedule (i.e., the same disk is idle in at least two successively-scheduled nodes). The second scheme ($S_2$) and the third scheme ($S_3$), on the other hand, are applicable when we have at least three and four consecutive “0”s in the schedule. Based on these power modes, the “Random” scheme can use $S_2$ for the third disk ($d_2$) and $S_3$ for the fourth disk ($d_3$). In comparison, the last column of the table on the right side of Figure 3.1 shows the result of the proposed scheduling that minimizes the Hamming distance between the successively-scheduled nodes, as explained above. We can see that this schedule is able to use scheme $S_1$ for disks $d_0$ and $d_1$, and scheme $S_3$ for disks $d_2$ and $d_3$, a much better situation compared to the random scheduling case. This small example illustrates how scheduling can impact the opportunities for disk power management.
An example IDAG constructed from PDAGs of processors $p_0$ and $p_1$.

(b) Scheduling obtained using the proposed algorithm.

(c) Another legal scheduling.

Figure 3.3. An example application of the proposed scheduling approach.

### 3.3.2 Multi-Processor Perspective

An IDAG is constructed from individual PDAGs. One potential problem with the single processor based approach explained in the previous section (that operates on individual PDAGs) is that the scheduling is performed for each processor independently. Consequently, while the resulting schedule can appear very good from the perspective of a given processor (as far as reducing disk energy is concerned), when IDAGs are considered together (i.e., the individual schedules are executed in parallel by observing data dependences across processors), they may not perform well. To illustrate this point, let us consider an IDAG for a two processor based system with 4 disks (see Figure 3.3(a)). As before, each node is annotated using its tag. Let us assume, for simplicity, each node takes the same time ($C$ cycles) to execute. In Figure 3.3(c), the columns titled as $p_0$ and $p_1$ give the schedules for the two processors (when each schedule is optimized independently as explained in Section 3.3.1). The last column (marked “Usage”), on the other hand, gives the disk usage when the interleaving effect of these two schedules are taken into account (i.e., each entry in the last column is the result of bit-wise OR of the corre-
sponding entries in the second and third columns). Under the same power-saving schemes assumed above (i.e., $S_1$, $S_2$, and $S_3$ in Figure 3.2), looking at the “Usage” column, we see that scheme $S_1$ can be used for disks $d_0$, $d_1$, and $d_2$, and there is no opportunity for applying $S_2$ or $S_3$. Figure 3.3(b) shows the result of the proposed scheduling. This scheduling, whose algorithm will be presented shortly, captures inter-processor effects and results in the disk usage shown in the last column of Figure 3.3(b). It can be observed that, in this case, we are able to use scheme $S_1$ for disks $d_0$, $d_2$, and $d_3$ and scheme $S_2$ for disk $d_1$. The proposed scheduling algorithm for an architecture with $P$ processors and $D$ disks is given in Figure 3.4. This algorithm takes an IDAG as input, and determines the schedule of nodes for each processor by considering the global (inter-processor) usage of the disks. It uses a $D$-bit global variable $G$ to represent the current usage of the disks. It schedules a node that is ready to be scheduled for each processor that finishes its current task. At each step, the algorithm first tries to schedule the node whose disk requirement can be satisfied with the current active disks, i.e., we can execute this node without requiring any disks currently in low-power mode. If multiple nodes satisfy this criterion, we select the one that requires the maximum number of disks to make full utilization of the currently active disks. If such a node does not exist, the proposed algorithm schedules the node whose tag is the closest (in terms of Hamming distance) to $G$, the bit pattern that represents the current disk usage (i.e., the disk usage at that particular point in scheduling). This is to minimize the number of disks whose (active/idle) states need to be changed.

### 3.3.3 Node Merging and Node Partitioning

In some cases, an IDAG may contain cycles which prevent a legal traversal (scheduling). We refer to these types of IDAGs as cyclic IDAGs. To schedule such graphs, we need to apply some node transformations and eliminate the cycles. An example cyclic IDAG is illustrated on the left side of Figure 3.5 for an I/O system with 4 disks. Notice that the nodes (classes) with tags 1101, 0001, and 0010 form a cycle, hence the IDAG shown in the figure is not scheduleable. We handle such graphs using two techniques, referred to as node merging and node partitioning in this
$$|a - b|$$ — Hamming distance between the class \( \{I\}_a \) and \( \{I\}_b \)

\text{last}[i] — the last class (node) executed on processor \( i \)

\text{schld}[i] — set of nodes already scheduled on processor \( i \)

\text{can_sch}[i] — set of nodes that can be scheduled on processor \( i \)

\text{next\_time}[i] — the time when the current node on processor \( i \) is finished

\( G \) — global disk usage bit vector

\( P \) — number of processors

\( D \) — number of disks

\( | \) — bit-wise “or”

\( \& \) — bit-wise “and”

\( \cup \) — unordered set union operation

\( \oplus \) — ordered set union operation (used for adding a node to set

\begin{verbatim}
for \( i := 0 \) to \( P - 1 \) do {
    \text{next\_time}[i] = 0;
    \text{last}[i] := 0;
}
\}
\end{verbatim}

\( G := 0; \)

while (exists unscheduled nodes) do {
    // determine \( S \) — the set of processors that are ready
    // to schedule new nodes
    \( t := \infty; \)
    for \( i := 0 \) to \( P - 1 \) do
        if (next\_time[i] < \( t \)) {
            \( S := \{i\}; \)
            \( t := \text{next\_time}[i]; \)
        } else if (next\_time[i] = \( t \))
            \( S := S \cup \{i\}; \)
    }

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

    // schedule nodes on the processors that have
    // finished with the previous nodes
    for each \( i \in S \) do {
        \( X := \{x|x \in \text{can\_sch}[i] \text{ and } (d[x] \& U) = d[x]\}; \)
        select \( x \in X \) such that \( |x - U| \) is minimized;
        // schedule \( x \) on \( P_i \);
        \text{schld}[i] := \text{schld}[i] \oplus x;
        \text{next\_time}[i] += t[x];
        \text{last}[i] := d[x];
    }

    // determine \( G \) — current usage of disks
    \( G := 0; \)
    for \( i := 0 \) to \( P \) do { \( G := G \mid \text{last}[i]; \) }

    for \( i := 0 \) to \( P \) do { update can\_sch[i]; } }

\end{verbatim}

**Figure 3.4.** Proposed scheduling algorithm with \( P \) processors and \( D \) disks.
Figure 3.5. An example that illustrates node merging and node partitioning.

The first transformation, node merging, combines all the nodes involved in a cycle into a single node. All incident edges on the nodes merged become incident edges on the combined node. The upper right part of Figure 3.5 illustrates how the nodes that form the cycle in this example can be merged, resulting in a scheduleable (acyclic) IDAG. The important point here is that node merging can be useful even when we do not have any cycles. This is because merging two nodes typically reduces the overhead to be incurred by the generated code and code expansion. One can see this by observing that the number of classes grows exponentially with respect to the number of disks. Therefore, if the underlying disks subsystem has too many disks, we may end up with too many nodes in the IDAG, and for each node we need to generate a different code (as will be explained in the following subsection). In such cases, reducing the number of nodes in the IDAG can be very useful since it reduces the size of the generated code and improves performance. However, a potential drawback of node merging is that the class that represents the combined node accesses in general more disks than the individual classes representing the nodes merged. More specifically, the tag of the combined node is the logical (bit-wise) OR of the tags of the constituent nodes. For example, in Figure 3.5, the tag of the resulting node is 1111, obtained by bit-wise ORing 1101, 0001, and 0010.

The other technique that can be used for eliminating a cycle in PDAG/IDAG

\footnote{An alternate approach would be constructing the IDAG in a cycle-free manner in the first place. We omit the detailed discussion of this alternative, since the results it generated were very similar to those obtained using node partitioning.}
is called node partitioning in this chapter. This transformation is in a sense the opposite of node merging, and generates multiple nodes from a single node. To illustrate how it operates, let us consider the original cyclic IDAG shown on the left side of Figure 3.5 again. The lower part of the same figure illustrates the acyclic IDAG obtained by partitioning the node with tag “1101”. It is assumed, for illustrative purposes, that after this partitioning, there is a dependence from node “1001” to one of the new nodes, and another dependence from node “0010” to the other new node. Notice that, in the worst case, each of these new nodes inherits the tag of the original node (as in the case of Figure 3.5). In general, the possibility of node partitioning can be checked as follows. Let us assume $I = I_{I_1} \cup I_{I_2} \cup \ldots \cup I_{I_n}$, where $I_{I_1}, I_{I_2}, \ldots, I_{I_n}$ are the nodes in a cycle. We select a node $I_{I_i}$ and split it into two nodes ($J_{I_i}$ and $K_{I_i}$) such that all the following constraints are satisfied:

\[ J_{I_i} \cap K_{I_i} = \emptyset \]  
\[ \{ J \rightarrow K | J \in J_{I_i}, K \in K_{I_i} \} = \emptyset \]  
\[ \{ J \rightarrow X | J \in J_{I_i}, X \in I - I_{I_i} \} = \emptyset \]  
\[ \{ X \rightarrow K | X \in I - I_{I_i}, K \in K_{I_i} \} = \emptyset \]

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

### 3.3.4 Implementation Details

This section gives details of how we generate scheduled code. The main issue here is to generate code for a given class ($I_{I_i}$). While one can propose an approach employing classical loop transformations such as loop tiling and loop interchange for this purpose, such an approach would not be sufficient, mainly because the iterations that belong to a class may not form a set that can be captured by these (structured) code transformations. Instead, in this study, we use a polyhedral
tool called the Omega Library to generate code. The Omega library [57] is a set of routines for manipulating linear constraints over integer variables, Presburger formulas, and integer tuple relations and sets. In our context, this library can be used for generating code that enumerates the iterations that belong to a given class. To see this, consider a scenario where a nested loop accesses the arrays stored in an I/O system that consists of two disks (each can hold 45 elements for illustrative purposes). Let us assume that there are two arrays accessed in the nest ($U$ and $V$), and that the array-to-disk mappings are as follows:

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

$$d_1 : \{ V[i] | 16 \leq i \leq 30 \}.$$

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

$$I_{p,10} = \{ i | (1 \leq i \leq 29) \land (1 \leq i \leq 30)$$
$$\land (1 \leq 31 - i \leq 15) \land \neg (16 \leq 31 - i \leq 30) \}.$$

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

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

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

$$I_{p,01} = \emptyset \quad \text{and} \quad I_{p,11} = \{ i | (1 \leq i \leq 15) \}.$$
3.3.5 Experiments

3.3.5.1 Setup and Benchmarks

To generate disk access patterns for our benchmark programs, we designed and implemented a trace generator. This trace generator creates a trace for each processor. The generated trace (which captures parallel disk accesses) is then fed to the simulator. The cycle estimates for the loop nests were obtained from the actual execution of the programs on a SUN Blade1000 machine (UltraSPARC-III architecture operating at 750 MHz with Solaris 2.9) and these estimates were used in all our simulations. In addition to the I/O trace file, the simulator needs the disk layout information for each array, which includes stripe unit size, striping factor (the number of disks), and starting disk. Using the disk layout parameters and traces, the simulator determines, for each request, the I/O node(s) that need to be accessed and the duration of access for each I/O node. We assume that each I/O node has one disk and no further striping is applied at the I/O node level, i.e., the data is striped across the I/O nodes only. In our simulator, the striping information is provided from an external file along with other simulation parameters. We used the same disk configuration used in the previous chapter and the default simulation parameters are given in Table 3.1.

Our disk power simulator, which is similar to DiskSim [41], is driven by the disk I/O request traces provided externally, which are generated, as explained above, by the trace generator. Each I/O request is composed of the following five parameters:

- The id of the processor that issues the request.
- Request arrival time: Time in ms specifying the time at which the disk request arrives.
- Block number: An integer specifying a logical disk block striped over several I/O nodes.
- Request size: An integer in bytes specifying the size of a request.
- Request type: A character specifying whether the request is a read (R) or a write (W).
Table 3.1. Default simulation parameters.

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

Table 3.2. Benchmarks and their characteristics.

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

Given an I/O trace file, the simulator generates statistical data for performance and energy consumption. Both performance and energy statistics were calculated based on the figures extracted from the data sheet of the IBM Ultrastar 36Z15 [2], and are given in Table 3.1. The values for power mode transitions are also included in Table 3.1. In the rest of the chapter, when we say “energy” we mean the energy consumed in the disk subsystem. When we say “execution time/cycles”, we mean the time/cycles it takes to complete the application execution. The disk energy consumption includes all the energy consumptions in both active and idle periods, taking into account all the states that the disks experience during the entire execution. Also, the performance numbers include all conflicts in accessing the parallel disk system.

Table 3.2 gives the set of array-based benchmark codes used in this study. These
benchmarks were randomly chosen from the SPEC2000 floating-point benchmark suite [42]. As none of the original SPEC 2000 requires memory footprint larger than 200MB, we increased the dataset size by manipulating the dimension sizes of the arrays and the corresponding loop bounds accordingly. We also made the data manipulated by these benchmarks disk-resident by mapping each array data to the corresponding file stored in the disk subsystem. As a result, each array reference causes a disk access unless the data is captured in the buffer cache. However, to be fair in our evaluation, we hand-optimized the I/O behavior of these applications as much as we could. In other words, even the original versions of these applications do not perform any unnecessary disk I/O. Also, to complete our simulations within a reasonable amount of time, we focused only on time-consuming loop nests from these applications. Specifically, from each application, we selected the loop nests whose cumulative I/O times account for at least 90% of the total I/O time of the application using the SUN Analyzer utility [58]. The second column in Table 3.2 gives the total disk-resident data size manipulated by the selected loop nests, and the third column shows the number of total disk requests made by each application. The last two columns, on the other hand, give the disk energy consumption and execution time, respectively, for each application when no disk power management is employed. The energy and performance numbers presented in the rest of this chapter are with respect to the values listed in these last two columns of Table 3.2.

### 3.3.5.2 Versions

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

- **Base**: This is the base version that does not employ any power management strategy. *All the reported disk energy and performance numbers are given as normalized values with respect to this version* (see the last two columns of Table 3.2).

- **TPM**: This is the traditional disk power management strategy used in studies such as [19] and [20]. In this approach, a disk is spun down after some idleness to save power, and is spun up when a new request arrives. Since the
performance cost of spinning up is typically large, TPM can incur significant performance degradations. Also, in order for this scheme to save power, the idleness should be large enough to compensate for the spin-up and spin-down latencies.

- **DRPM**: This is the dynamic RPM strategy proposed in [22]. Considering the predicted length of the idleness, it sets the rotation speed of the disk to an appropriate level to save power. Therefore, it is effective in saving power even if the idle periods are short. Note that the RPM level used is selected based on the estimated idleness (as in [22]) and we may incur performance penalties, depending on the accuracy of idle time prediction.

- **Compiler-directed TPM (C-TPM)**: This proactive scheme lets the compiler estimate idle periods by analyzing code and considering disk layouts, and then generates the necessary TPM power-management calls (spin\_down and spin\_up calls) based on this information.

- **Compiler-directed DRPM (C-DRPM)**: This proactive scheme performs the same estimation of idle periods as in C-TPM, but it generates explicit DRPM power-management calls (set\_speed calls). Both C-TPM and C-DRPM are discussed in more detail in Section 2.4.2.

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

- **Intra-processor DRPM (Intra-P-DRPM)**: This corresponds to our code restructuring based approach (from a single-processor perspective) when it is used with C-DRPM. It uses the same (restructured) code as in Intra-P-TPM. The scheduling strategy used by Intra-P-TPM and Intra-P-DRPM are explained in Section 3.3.1.

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

- **Inter-processor DRPM (Inter-P-DRPM):** This corresponds to our code restructuring based approach (from a multi-processor perspective) when it is used with C-DRPM. It uses the same restructured code as in Inter-P-TPM. The scheduling strategy used by Intra-P-TPM and Inter-P-DRPM are explained in Section 3.3.2.

Note that the only modification to the input code made by C-TPM and C-DRPM are insertions of the explicit disk power management calls, which are then simulated by the disk simulator. In comparison, the Intra-P-TPM, Intra-P-DRPM, Inter-P-TPM and Inter-P-DRPM schemes restructure the application code using scheduling. The necessary code modifications for these schemes are automated using the SUIF infrastructure [43], with the help of Omega Library [57] as has been discussed earlier. As a result of these compiler transformations, we observed that the original compilation times were almost doubled. We believe that, considering the large benefits of the approach, this increase in compilation times is tolerable.

### 3.3.5.3 Experimental Results

The graph in Figure 3.6 gives the energy consumptions of our benchmarks under the different schemes explained above. One can make several observations from these results. First, the TPM scheme does not achieve any disk energy savings since most of the disk idle times in these applications are not very large as shown in Figure 3.8. And, for very few relatively long idle periods, the TPM scheme fails to exploit them as well, mainly because it waits for some time (at the beginning of each idle period) before spinning down the disk (see Figure 2.5). In comparison, C-TPM brings about 9% energy savings by taking advantage of these few relatively long idle periods, which demonstrates the benefits brought by proactive disk power management. The second observation is that the DRPM scheme consumes more energy than the original version (Base), due to poor estimation of idle periods. However, its proactive version (C-DRPM) achieves nearly 24% disk energy savings on average. Our third observation is that the best results for all applications are obtained with the Inter-P-TPM and Inter-P-DRPM versions. Specifically, they
Figure 3.6. Energy consumptions with different schemes.

Figure 3.7. Execution cycles with different schemes.

Figure 3.8. CDF (cumulative distribution function) curves for disk idle times. An (x,y) point on a curve indicates that y% of the idle times has a duration of x (ms) or lower. As mentioned earlier, the minimum amount of idle time required to compensate the cost of spinning down the disk and up (under a TPM-based scheme) is called the threshold. Based on the numbers from IBM Ultrastar 36Z15, the threshold is 15.19 seconds. The results in this graph show that the idle disk times exhibited by these array-based applications are much shorter than the threshold value.

achieve, respectively, about 38% and 43% savings in disk energy. In comparison, Intra-P-TPM and Intra-P-DRPM save approximately 18% and 30% disk energy, respectively. In other words, capturing and exploiting interprocessor disk access pattern is critical in maximizing savings. In fact, Inter-P-TPM generates better energy savings on average than Intra-P-DRPM, meaning that using a less powerful architectural mechanism with more sophisticated code restructuring generates better results than employing more powerful architectural mechanism with less sophisticated code restructuring for this set of applications.
It is to be noted however that the energy consumption is just one part of the big picture. To have a fair comparison between the different schemes tested, one needs to consider their performances (i.e., execution times/cycles) as well. The bar-chart in Figure 3.7 gives the normalized execution times (with respect to the base version) for the different schemes evaluated. One can observe that only the DRPM version incurs some performance penalty, 70% on average across our four benchmarks. The reason why TPM does not incur any performance penalty is that it is not generally applicable, given the short disk idle times as discussed earlier. We also see that all the compiler-directed schemes, C-DRPM, C-TPM, Intra-P-TPM, Intra-P-DRPM, Inter-P-TPM, and Inter-P-DRPM, incur almost no performance penalty. The main reason for this is that these schemes start to bring the disk to the desired RPM level before it is actually needed (using pre-activation), and the disk becomes ready when the access takes place. This is achieved by accurate prediction of the disk idle periods for the application domain we target. These results, along with those presented in Figure 3.6, indicate that the compiler-guided proactive disk power management and code restructuring can be very useful in practice, in terms of both disk energy consumption and execution time penalty, and the best savings are achieved by our code restructuring approach. Note that, since our compiler approach does not increase execution times, it does not cause much extra power consumption on other system components. The only additional energy overhead is due to execution of the inserted power management calls (instructions); but, we found this cost to be negligible.
In the rest of our experimental analysis, we vary the values of some of the simulation parameters, and study their impacts on energy consumption. We do not present any further performance data, mainly because, except for the DRPM scheme, none of the schemes evaluated causes any substantial increase in original execution cycles. More specifically, except for DRPM, the average execution time increase was always less than 1%. We focus on two important parameters in our sensitivity analysis: disk layout and number of processors. Since disk layout has three components as explained in Section 2.4.1, we study each of them separately. In each experiment, we change the value of only one parameter; the rest of the simulation parameters use their default values listed in Table 3.1. Also, since our results with different benchmarks resulted in similar trends and observations, we present the result for the mgrid benchmark only.

Figure 3.9 gives the normalized energy consumptions with the different stripe sizes. We see from these results that the energy savings brought by the compiler-based schemes increase as the stripe size increases. This can be explained as follows. When the stripe size is very small (16K), disks do not experience much idleness. In fact, the disk idleness in this case becomes so small that even code restructuring cannot take much advantage of it. When the stripe size is increased, more disk requests can be serviced by a single stripe, i.e., the stripe-level data reuse improves. As a result, the compiler-based approaches have more opportunities for optimization, which in turn helps reduce disk energy consumption. We see from Figure 3.9 that Inter-P-DRPM generates the best savings with 256K stripe size, and the difference between it and the DRPM scheme reaches its peak at this value. The next parameter whose variation we study is the stripe factor (the number of disks). Recall that the default number of disks used so far in our experiments was 8. Figure 3.10 gives the normalized energy values under different stripe factors. An observation we can make from these curves is that, when we have only two disks, there is not much opportunity for power saving (due to lack of disk idleness), and (except for DRPM) all the schemes behave similarly. As the number of disks is increased, disk idleness increases, and consequently, the compiler schemes exhibit a better behavior. When the number of disks is very high (32), the disk idleness reaches a very high level, and one may not need sophisticated code restructuring in this case (for our particular data set sizes). In fact, at this point, all the
TPM-based compiler schemes behave similarly, and all the DRPM-based compiler schemes behave similarly.

We next study how the starting disk used for striping could affect the results. To perform this set of experiments, we generated a random integer number (for each array in the mgrid benchmark) between 1 and 8 to select the disk from which the array is striped. The results for five such experiments are presented in Figure 3.11. We see from these results that the general trends (and our savings) are very similar across these different layouts. This indicates that the starting disk (for striping) may not be a very important factor as far as the impact of our compiler-based schemes are concerned.

The next parameter whose variation we study is the number of processors. Figures 3.12 gives the normalized energy results with the different processor counts. As before, all other parameters are set to their default values given in Table 3.1. One can see from these results that the effectiveness of the compiler-directed code restructuring is consistent across the different processor counts. The reason that the C-TPM and C-DRPM schemes do not behave very well with the large number of processors is the difficulty in inserting explicit power management calls, due to small iteration counts with large number of processors. This problem does not usually exist in the code restructuring based schemes since they cluster idle and active periods.

In our next set of experiments, we conduct an experiment with up to 128 disk to see how the energy savings we achieved are affected with the larger number.
of disks. We also increased the problem size as we deal with the larger amount of data. Figure 3.13 shows the normalized energy consumption results with the different number of disks, from 8 to 128. As we can see, in most cases, the compiler directed schemes achieve better energy savings than the hardware based schemes, such as TPM and DRPM. One noticeable trend that can be observed is that the TPM scheme increases energy consumption dramatically with the larger number of disks. The main reason for this is that TPM works in a reactive manner. That is, even if the disk can have more chances to spin down due to the increase in disk idle time with the increased number of disks, it also needs to pay spin-up cost, which incurs both energy and performance penalties. Another observation we can make from these results is that, as the number of disks increases, the TPM flavored compiler schemes exhibit better energy savings than the DRPM flavored ones. This is because spinning down to shutdown mode consumes much less energy than modulating the disk in a lower RPM, even if it is the lowest one.

Figure 3.13. Scalability of various schemes experimented.
3.4 Loop Transformation

This section presents how two conventional loop transformations, loop distribution and loop tiling, can be applied to reduce disk energy consumption by changing the disk access pattern.

3.4.1 Loop Distribution

To illustrate how a loop transformation can be useful in reducing disk energy, let us first consider the code fragment shown in Figure 3.14(a) that manipulates four disk-resident arrays (U1, U2, U3, and U4). Let us assume, for illustrative purposes, that we have a total of 8 disks, and arrays U1 and U2 are striped over the first 4 disks, whereas arrays U3 and U4 are striped over the other 4 disks, as shown in Figure 3.14(c). Assume further that the disks in the system are equipped with the DRPM capability. This code fragment is unlikely to take any advantage of DRPM since the idleness duration for each disk is expected to be too short. Figure 3.14(b) gives the transformed code after loop distribution [23], a transformation typically used for enhancing loop-level parallelism and instruction cache performance. Note that what this transformation does in this example is to put the two statements in the original loop body (S1 and S2) into separate loops. As a consequence, now, during the execution of the first loop, only the first 4 disks are used, and the remaining disks can be placed into the low-power mode (i.e., they can reduce their speed or spin down). Similarly, during the execution of the second loop, only the last 4 disks are accessed, and the first 4 disks can be put in the low-power mode. This small example illustrates how a code transformation such as loop distribution (also called loop fission) can improve the effectiveness of disk power management. In this example, the idleness generated by the transformation can be large enough to accommodate even TPM, which allows us to use TPM a viable option for array-based codes.

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. For instance, in our current example, if all the four arrays were striped over all eight disks available, loop
Figure 3.14. Loop fission example. (a) Original code fragment. (b) Transformed code fragment. (c) Disk layout for the arrays.

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 3.15. Loop fissioning algorithm.

Figure 3.16. Example application of the proposed loop fission algorithm. (a) Original code fragment. (b) Transformed code fragment. (c) Disk allocation for array groups.
distribution would not be of much use, since it would not help increase the disk inter-access times significantly (depending on the stripe size and the loop iteration counts). This discussion suggest that, unlike the case with the conventional compilation frameworks that target data cache locality, in this 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 2.4.1, 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 3.15 determines the necessary loop distribution (for each nest) and the corresponding 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 3.16 illustrates an example application of this algorithm.

In the original code fragment shown in Figure 3.16(a), three loop nests access a total of 10 arrays ($U_1$, $U_2$, $U_3$, $U_4$, $U_5$). Assuming that all the arrays are of the same size and each loop nest has the same iteration count, the proposed approach forms four array groups, $\{U_1, U_2, U_5\}$, $\{U_3, U_4, U_8\}$, $\{U_6, U_7\}$, and $\{U_9, U_{10}\}$. Note that, $U_2$ and $U_5$ belong to the same group, as they are coupled via array $U_1$. These four array groups are assigned to the disks as shown in Figure 3.16(c). The fissioned code is given in Figure 3.16(b). Now, for example, when the application executes the first loop in the transformed code, the execution accesses only the first three
INPUT: A loop nest and disk layout information.
OUTPUT: An energy-optimized tiled code and the transformed disk layout.

Begin
\[ N \leftarrow \text{the number of arrays accessed by the nest} \]
Create tiled loop nest with tile size, \( TS \)
\begin{itemize}
  \item for \( i = 1 \) to \( N \) do
    \begin{itemize}
      \item Determine the data size, \( DS(i) \), required while iterating given \( TS \).
    \end{itemize}
  \end{itemize}
\begin{itemize}
  \item for each array used in the loop nest do
    \begin{itemize}
      \item if data access pattern \( \neq \) storage pattern then
        \begin{itemize}
          \item Transform the data layout (e.g., from row-major to column-major).
        \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}
Reshape access patterns of arrays.
\begin{itemize}
  \item for \( i = 1 \) to \( N \) do
    \begin{itemize}
      \item \text{stripe size}(i) \leftarrow DS(i) \text{ of each array}
    \end{itemize}
  \end{itemize}
End

Figure 3.17. Loop tiling algorithm.

disks; one can save a significant amount of energy by putting the unused disks in the low-power mode.

3.4.2 Loop Tiling

Another loop-based transformation, loop tiling [59], 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 tile. Therefore, if done appropriately, for a given execution of the tile iterators, the element iterators access only specific blocks of data. In this 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 the layout-aware loop tiling algorithm for reducing disk energy consumption is given in Figure 3.17. An
Figure 3.18. 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.

example application of this algorithm is illustrated in Figure 3.18. Figure 3.18(a) shows the original code fragment, and Figure 3.18(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 3.18(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 3.18(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.

3.4.3 Experiments

To evaluate the impact of two loop transformations, i.e., loop distribution and loop tiling, and that of the disk layout optimization 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 5.2.

- **TL+DL**: The layout-aware tiling version based on the algorithm given in Figure 3.17.

It is to be noted that, each of these versions can be combined with TPM, ITPM, DRPM, IDRPM, CMTPM, or CMDRPM as presented in Chapter 2 (see Section 2.5). That is, the proposed code (and disk 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, DL captures different meanings 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 disk layout transformation and tile-to-disk mapping. We also want to mention that, since the loop transformation changes the original loop structure and consequently disk access patterns as well, we obtained the energy and performance values from the disk traces using the transformed code execution.

The normalized energy consumption results for the several SPECfp2000 benchmarks under the different schemes are given in Figure 3.19. All the results are normalized with respect to the base version that does not employ any power saving techniques. 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 whether data is stored in row-major or column-major order, and indiscriminately fissioning or tiling the loops without considering the disk layouts of the arrays involved does not lengthen disk inter-access times. We also observe that the LF and TL versions increase the number of loop nest or the depth of nest, which results in a slight performance degradation due to the increased loop control overhead. The mere LF and TL versions therefore have little impact on reducing the disk energy consumption. In other words, as mentioned earlier, loop distribution and loop tiling make sense in our context only if they are accompanied with a suitable disk 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 array storage order. So, after transforming the array storage order 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 array storage order. Maybe the most interesting trend that can be observed from Figure 3.19 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 such opportunities for it, and consequently it reduces the energy consumption of the base case by 31%, on average. Overall, these results show that the layout-aware code transformations and disk layout optimization can be very useful in practice. 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.
3.5 Code Restructuring for Exploiting Disk Access Locality

3.5.1 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., [60, 61]), 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 execution, 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 [62]. Note that, in this section, 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 [62]. 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:

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

Considering (only) this distance vector, the $k$th loop can be parallelized if at least one of the two conditions below are satisfied [62]:

$$d_k > 0 \quad \text{or} \quad d_k = 0.$$
Figure 3.20. An example data access pattern scenario that involves four processors. In this scenario, three different loop nests access the same array.

- \( d_k = 0 \)
- \((d_1 \ d_2 \ \cdots \ d_{k-1})^T\) 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 \( k \)th 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 3.20, 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 iteration space of each nest. The array (data space) manipulated by these loop nests (note that all three processors manipulate the same
Figure 3.21. Two different parallelizations from the perspective of a given processor. (a) Loop-based parallelization. (b) Reuse-aware parallelization.

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 3.21(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.

3.5.2 Disk Layout-Aware Code Parallelization

Figure 3.21(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 3.21(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 3.21(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 3.5) can be applied to the code of each processor separately to further increase disk idleness (and energy savings).

### 3.5.2.1 Mathematical Details

We now discuss the details of the mathematical engine behind the proposed approach to disk reuse in the multiprocessor case. The proposed 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 \( Q_k \) (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 \( Z_j \) to represent the set of data items in \( U_j \). Note that \( Z_j \) 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 the proposed 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 \( Z_{s,j} \) 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 \( Q_{s,j,k} \) to represent the set of loop iterations from loop nest \( k \) that touch the elements in \( Z_{s,j} \). The proposed parallelization strategy assigns the iterations in \( Q_{s,j,k} \) 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 $Q_{s,j,1}$, $Q_{s,j,2}$, ..., $Q_{s,j,n}$. The common characteristic of these sets is that all the iterations in them access $Z_{s,j}$. Consequently, when all the iteration assignments (for all loop nests) are complete, we have the scenario shown in Figure 3.21(b) for the example in Figure 3.20.

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 the 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. The proposed 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 [63]. This approach implements a method for deriving an optimal hyper-parallelepiped 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 inter-processor data communication traffic based on a data footprint concept. After parallelizing the code using the approach in [63], for each loop nest, the proposed 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 $I_s$ is the set of iterations assigned to processor $s$ (by the loop parallelization used) and $D_s$ is the set of array elements we want to determine:

$$D_s = \{ \vec{d} \mid \exists I \in I_s, \exists R \in R_s \text{ such that } R(I) = \vec{d} \}.$$  

In this formulation, $R_s$ is the set of references to the disk-resident array and
that are assigned to processor resident array $U$

Basically, using the approach in [63], we first determine the sets $Q_{s,j}$, then determine $Z_{s,j}$, and finally determine $D_{s}$. 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).

$R$ 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 the proposed 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 $D_{s_1}$, $D_{s_2}$ and $D_{s_3}$ 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 this, it selects the row-block distribution ($D_{s_1}$), as it is requested by a larger number of processors; i.e., we set $Z_{s,j}$ to $D_{s_1}$. Figure 3.22 summarizes the process of determining the loop iterations that will be executed by processor $s$. Basically, using the approach in [63], we first determine the sets $D_{s_1}$, $D_{s_2}$, ..., $D_{s_n}$. We next obtain $Z_{s,j}$ using the strategy explained in the previous paragraph, and then determine $Q_{s,j,1}$, $Q_{s,j,2}$, ..., $Q_{s,j,n}$, 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 $M'_{j,k}$ and $M'_{j,l}$ to denote the set of elements (of the disk-resident array $U_j$) accessed by nests $k$ and $l$, respectively (note that $M'_{j,k} \subseteq Z_j$.
and $M'_{j,l} \subseteq Z_j$). We can express the problem as follows: Divide the iterations in $Q_k$ and $Q_l$ across the $p$ processors such that the parts assigned to processor $s$ from these two nests, namely, $Q_{s,j,k}$ and $Q_{s,j,l}$, access the same set of elements as much as possible. In mathematical terms, the proposed approach proceeds as follows. We first determine the set of common elements between $M'_{j,k}$ and $M'_{j,l}$, i.e., $M'_{j,\text{common}} = M'_{j,k} \cap M'_{j,l}$. Then, we assign the first $|M'_{j,k}|/p$ of these ($|M'_{j,\text{common}}|$) elements to the first processor, the next $|M'_{j,k}|/p$ to the second processor, and so on. At the point where we have assigned all ($|M'_{j,\text{common}}|$) elements, the remaining elements (i.e., $|M'_{j,k}| - |M'_{j,\text{common}}|$) 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.

The proposed 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
    ... U_1[i][j] ... U_2[j][i] ... U_3[i+2][j-3] ...
```

We note that, for a given loop iteration $(a, b)$ in this nest, 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 $A_{s,1,k}$, $A_{s,2,k}$, $A_{s,3,k}$, ..., $A_{s,m,k}$ 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 $Z_{s,j}$, the proposed approach uses these regions. After computing
Figure 3.23. 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).

$Z_{s,j}$, the sets $Q_{s,j,1}, Q_{s,j,2}, \ldots, Q_{s,j,n}$, i.e., the iterations from the different nests that are assigned to processor $s$, can be computed as has been discussed earlier. Figure 3.23 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 dissertation considers \textit{all} the loop nests at the same time, and in general, the output (restructured) code generated by the proposed approach cannot be obtained by simple loop fusion.

Figure 3.24 shows an example application of our disk layout-aware code parallelization. Figure 3.24(a) shows a code fragment that consists of three loop nests: $L_1$, $L_2$, and $L_3$. All these loop nests access array $U_1$. Let us assume we have four processors. By using the approach presented in [63], we partition each loop nest into four parts, and Figure 3.24(b) shows the loop nests executed on processor $s$ (where $s = 1, 2, 3, 4$). The right side of Figure 3.24(b) gives the partitioning of array $U_1$ demanded by each loop nest. As we can observe, loop nests $L_1$ and $L_2$ demand array $U_1$ be partitioned in column-blocks, while loop nest $L_3$ demands this array be partitioned in row-blocks. Since column-block is the most popularly demanded partitioning, we determine to partition array $U_1$ into column-blocks, as shown in Figure 3.24(c).\footnote{Instead of employing majority voting across the nests, we can also apply weighted majority voting where the number of loop iterations of each nest is taken into account. In our application}
Figure 3.24. Example application of our disk layout-aware code parallelization.

tribute the iterations of each loop nest shown in Figure 3.24(a). Figure 3.24(d) shows the parallelized loop nests scheduled to be executed on processor $s$. Note that, in the transformed code, a given processor accesses the same set of array elements in each loop nest. Consequently, each processor will exercise the same set of disks in each loop nest, which is expected to lead to good disk reuse, and eventually to better energy saving.

programs, simple majority voting and weighted majority voting did not make significant difference
Table 3.3. Applications and their characteristics.

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

3.5.3 Experiments

3.5.3.1 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 determines 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. We used the same disk configuration used in the experiments given in Section 3.3 and 3.4, and the default simulation parameters are given in Table 3.1.

The simulator we use, which is similar to DiskSim [41], 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 five 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 [2], and are given in Table 3.1. The values for power mode transitions are also included in Table 3.1. As to the power model of DRPM disks, we obtained these values using quadratic estimation described in [22].

Table 3.3 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 3.3 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 section are normalized with respect to the values listed in these last two columns of Table 3.3. 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 3.3.

• **TPM**: This is the traditional disk power management strategy used in studies such as [19] and [20]. 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 sig-
nificant 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 [22].
  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 ap-
  proach when it is used with TPM. As has been discussed in Chapter 3.2, 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 ex-
  ploits disk layout information to restructure the given application code (see
  Section 3.3).

- **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 3.5.

- **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 3.5.

### 3.5.3.2 Results

The graph in Figure 3.25 gives the energy consumption of our applications under
the different schemes discussed earlier when we execute them on a single proc-
essor. One can make several observations from these results. First, the TPM version
Figure 3.25. Normalized energy consumption results with the single processor execution.

Figure 3.26. Normalized energy consumption results with 4 processor execution.

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, as illustrated in Figure 3.27. In comparison, the DRPM version generates better results, achieving an average disk energy saving of 9.95%. These results are consistent with those found by the prior research [22], 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 small 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. To better explain why this scheme save energy, we plot in Figure 3.28 the CDF curves of disk idle times when this scheme is used. Comparing this plot to the one in Figure 3.27 clearly shows that our approach increases idle periods significantly.

Figure 3.26 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
Figure 3.27. CDF (cumulative distribution function) curves for disk idle times when TPM is used. An (x,y) point on a curve indicates that y% of disk idle times has a duration of x (ms) or lower.

Figure 3.28. CDF curves for disk idle times when T-DRPM-s is used.

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 3.29. 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 3.3). 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 (see Figure 3.27). 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 3.30). 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-charts in Figures 3.25, 3.26, 3.29, and 3.30 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.

In the rest of our experimental evaluation, we modify the default values of some of our simulation parameters and conduct a sensitivity analysis. In each experiment, we change the value of only one parameter; the values of the rest of the parameters are kept at their defaults in Table 3.1 to make it easy to interpret the results observed. Also, we present only the energy results for the 4 processor case; the performance results are omitted as they are similar to those in Figure 3.30 in all the experiments. Since the trends with the different applications are very similar to each other, we present the results only for the FFT application. The
The first parameter whose value we study is the number of disks used for striping data arrays (stripe factor). Figure 3.31 presents the normalized energy results with the different stripe factors. We observe an increase in the savings achieved by our schemes (T-TPM-* and T-DRPM-*) for all the stripe factors tested. This is because using a large number of disks gives more opportunities to our approach for maximizing the number of idle disks at any given time and the underlying power management mechanism takes advantage of this.

The second parameter we study is the start disk for striping. Recall that the default value for this parameter used in the experiments so far was 1 (i.e., the first disk in the system). The energy results with the different start disks (when stripe factor is set to 8) for different arrays are presented in Figure 3.32.
specifically, in this experiment, each of the six disk-resident arrays in the FFT is started from a different disk. For ease of comparison, we also reproduce the results from Figure 3.26. One can see from these results that the trends do not change significantly when the start disks for arrays are changed. The last parameter whose value we vary is the stripe size. Recall from Table 3.1 that the default stripe size used in the experiments so far was 32KB. The results with the different stripe sizes are given in Figure 3.33. The effectiveness of our versions increases as we increase the stripe size. The reason for this behavior is that a larger stripe size allows us to schedule a large number of loop iterations when processing a disk, and this tends to increase idleness of the remaining disks in the system. To sum up, considering the trends captured in Figures 3.31, 3.32 and 3.33 together, we can conclude that our approach is effective across a wide range of values of the disk layout parameters.

### 3.6 Summary

To increase the effectiveness of underlying disk power management schemes, it is very crucial to enlarge disk idle periods. This chapter presents how compiler-based code transformation and restructuring improve the effectiveness of underlying disk power management schemes. More specifically, we first present a code scheduling technique that allows more effective disk power management. We also present 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. Lastly, we
present a scheme that restructures 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.
Chapter 4

Energy-Aware Prefetching

4.1 Introduction

While conventional disk power optimization approach [20, 19, 21] based on spinning down idle disks has been successful in the context of laptop disks, it is not the best option for server disks and scientific workloads that exhibit very short idle disk periods. Therefore, one of the prior proposals [10, 22] to disk power saving in high-performance systems has been to employ disks with the capability of changing their rotational speeds dynamically. Since such multi-speed disks (e.g., those from [37], [38], [39]) can serve requests even under low rotational speeds, they can potentially exploit short idle periods as well and, at the same time, save power (due to reduced speed). However, the question of whether one can increase the power savings that could be achieved through such multi-speed disks remains important and largely unexplored. In particular, the role of the software-level optimizations for utilizing such multi-speed disks in the most effective way needs to be investigated.

The goal of this chapter is to demonstrate that compiler-directed rescheduling of disk access instructions in scientific applications can be very effective in practice and increase power savings obtained from multi-speed disks significantly. The specific strategy proposed and evaluated in this work “hoists” disk access instructions in the program code to increase the time-gap between the issue of the instruction and the actual access to the disk. In this way, the hoisted instruction can use a disk that operates with a lower speed than the maximum one. More specifically, the approach proposed in this chapter determines the most suitable prefetch distance
for each array reference in the application code, disk speeds (RPM levels) for all
the disks in the storage system, and data layouts for the disk-resident arrays in
a unified setting. Note that since our goal is to issue prefetches to disks that
rotate at lower speeds, our prefetch distances are larger than those normally used
in conventional I/O prefetching.

4.2 Related Work

I/O prefetching has been known to be a very effective way of improving I/O per-
formance. The basic idea is to hide I/O stall time by issuing I/O commands ahead
of the time when they are actually needed. Since prefetched blocks can pollute the
cache for normal cached blocks, prefetching should be designed and implemented
carefully. Mowry et al [64] used compiler-generated information to manage prefetch
commands more effectively in the context of single CPU execution. They also stud-
ied the cases where processes running concurrently on the same CPU generate I/O
prefetch commands simultaneously [65]. To handle the interaction among multiple
I/O prefetch instructions coming from multiple processes, they also introduced a
release command in addition to compiler-inserted prefetch command [65]. Li and
Shen [66] proposed a memory management framework that handles non-accessed
but prefetched pages separately from the rest of the memory buffer cache. They
employ different heuristic policies to select a victim page. They either evict the
last page of the longest prefetch stream, or the last page of the least recently ac-
cessed prefetch stream, or evict the last page of the prefetch stream whose owner
process has consumed the most amount of CPU since it last accessed the prefetch
stream. For sequential workloads, Gill and Modha proposed a novel prefetching
scheme, called SARC [67], that dynamically adapts the cache space between se-
quential and random streams. In TIP (Transparent Informed Prefetching) [68],
“prefetch horizon” is used to limit the prefetch depth, beyond which there is no
benefit from prefetching. While prefetch horizon can mitigate the impact of aggres-
sive prefetching on effective cache size, this approach does not specifically target
harmful prefetches. Patterson et al also studied the same problem under multi-
process execution environments [69]. More recently, Ding et al proposed a scheme,
called DiskSeen [70], that improves I/O prefetching by exploiting disk layout and
access patterns observed to overcome the inherent limitations of prefetching at the logical file level. Unlike these prior studies that considered I/O prefetching from the aspect of performance only, our approach described in this chapter is different from these, since our main goal is to reduce energy consumption by disk subsystem by employing I/O prefetching in an energy-aware fashion.

4.3 Background and Motivation

In this section, we present how we can reduce disk energy consumption by hiding latencies of low-speed disks using the example code fragment shown in Figure 4.1(a). The code fragment given in this figure accesses a two-dimensional disk-resident array, named $V_1$, using a loop nest constructed from two loops. For illustrative purposes, $V_1$ is assumed to be striped over 4 disks with a stripe size of $S$ (see Figure 4.1(b)) and all four disks in question are assumed to be running at 12,000 RPMs. As depicted in Figure 4.2(a), if we do not apply any prefetching, every access to the first data element in each block incurs an access ($R_i$) to the disk system. In this example, let us assume that it takes $T_d$ cycles to complete a disk access when the rotational speed of disks is 12,000 RPM. After $T_d$ cycles elapse, the requested data block is ready ($D_i$) and thus the computation on that block can proceed.

As this dissertation targets at scientific benchmarks whose access patterns can be extracted and reshaped by an optimizing compiler, we can use the software prefetching algorithm proposed by Brown et al [18] to hide disk I/O stall time and reduce overall execution latency. The code fragment after applying I/O prefetching is given in Figure 4.1(c). Software prefetching generates a prolog, a steady-state, and an epilog from each original loop nest. The prefetch distance ($d$), i.e., the number of iterations ahead of which the disk I/O needs to be initiated to hide I/O latency, can be calculated as:

$$d = \lceil \frac{T_d}{s + T_{pf}} \rceil,$$

where $T_d$ is the estimated I/O latency (in cycles) to prefetch one block, $T_{pf}$ is the overhead (again in terms of cycles) of executing a prefetch instruction, and
Figure 4.1. An example application of prefetching.

$s$ is the number of cycles in the shortest path through the loop body. Once the prefetch distance, $d$, is calculated, we then stripe-mine the loop nest to make explicit the point at which the prefetch instruction is to be inserted. The result of this transformation for our example is given in Figure 4.1(c). In this example, $d$ iterations of $j$ loop are assumed to be required to hide I/O latency and $b_1$ is the strip size used for strip-mining.

Up to this point, we discussed software prefetching as a technique that can be used to hide disk I/O latency, specifically hiding $T_d$, as proposed in the literature. However, if we examine the components of disk I/O time, we can see that $T_d$ is composed of seek time, rotational latency, transfer time, and controller overhead. Since in modern disk drives the controller overhead is negligible compared to other three values, we can see that $T_d$ is almost directly proportional to the disk rotation speed. However, it has been shown by prior research that the disk power consumption is quadratically proportional to the disk rotational speed [22]. This suggests that one can take an approach to conserve disk energy by storing array data in low-speed disks, e.g., a disk running at lower than 12,000 RPM (in this example), and by eliminating the increased I/O latency using software prefetching.
with an increased prefetch distance. That is, one can save disk energy by increasing prefetch distance and reducing disk speed at the same time.

Figure 4.2 and Figure 4.3 show how prefetching to high-speed disks and low-speed disks affect I/O timing and disk power consumption. In this example, the rotational speed of the low-speed disks is assumed to be 6,000 RPM (i.e., half of the maximum speed possible). Consequently, the time it takes to complete a disk access is doubled, i.e., it is now $2T_d$. One can see from Figures 4.2(b) and (c) that we can hide the latency of low-speed disks by issuing the prefetch early enough. Specifically, since the I/O latency is doubled from $T_d$ to $2T_d$, the prefetch distance ($d$) is also doubled based on Equation (4.1) given above. On the other hand, the energy consumption profiles after applying prefetching with different prefetch distances are depicted in Figure 4.3. Figure 4.3(a) shows the power profile throughout the program execution time when no prefetching is employed. Note that we assume the disk drive can be placed in either active mode when servicing I/O request or idle mode when the disk is not used. Therefore, the disk is in the active mode during $T_d$ when there is a request being processed. For the remaining time, the disk is placed into the idle mode. Figures 4.3(b) and (c) show how the prefetching affects the power consumption profile of a disk. If we apply prefetching using high-speed disks, we can conserve disk energy consumption by the amount of reduced execution time. In this case, the energy savings come from the reductions in the total disk idle time. In comparison, as shown in Figure 4.3(c), if the data is stored in low-speed disks and we apply prefetching, we can reduce disk energy consumption further by cutting the energy consumption in the active and idle periods as well.

It should be noted that we may not be able to take advantage of low-speed disks for all disk-resident arrays due to following reasons: As mentioned earlier in this section, using low-speed disks entails longer prefetch distances, which may not be very appropriate for a loop nest whose iteration count is not sufficient for hiding such a long I/O latency. Therefore, one needs to be careful when selecting the disk speeds to employ. Also, since we focus on large scientific programs that consist of multiple loop nests, it is possible that the determined disk speed for a particular array in one loop nest may not be appropriate for another loop nest that manipulates the same array (by accessing the same set of disks). Consequently,
selecting prefetching distance and disk speeds depends on the disk layout of data as well as the data access patterns exhibited by the application code being optimized. Because of this, these parameters should be considered together.
4.4 Energy-Aware Prefetching Algorithm

4.4.1 Basics

Before describing the algorithm, let us first define a few important mathematical concepts. In the proposed approach, an array based/loop-intensive program \( P \) that consists of \( s \) loop nests is represented as:

\[
P = (L_1, L_2, \ldots, L_s),
\]

(4.2)

where \( L_i(i = 1, 2, \ldots, s) \) is the \( i \)th loop nest in program \( P \). We further assume that a loop nest \( L_i \) of the following form:

\[
L_i: \text{for } i_1 = l_1 \text{ to } u_1, \text{ step } b_1 \\
\quad \text{for } i_2 = l_2 \text{ to } u_2, \text{ step } b_2 \\
\quad \ldots \\
\quad \text{for } i_k = l_k \text{ to } u_k, \text{ step } b_k \\
\quad \{\text{loop body}\}
\]


can be represented as:

\[
L_i = \text{for } \vec{I} \in [\vec{L}_i, \vec{U}_i], \text{ step } \vec{b} \ \langle a_1(\vec{I}), a_2(\vec{I}), \ldots, a_m(\vec{I}) \rangle,
\]

(4.3)

where \( \vec{I} \) is the iteration vector, and \( \vec{L} = (l_1, l_2, \ldots, l_k)^T \) and \( \vec{U} = (u_1, u_2, \ldots, u_k)^T \) are the lower and upper bound vectors, \( \vec{b} = (b_1, b_2, \ldots, b_k)^T \) is the loop step vector, and \( a_j(\vec{I}) \) \((j = 1, 2, \ldots, m)\) is the \( j \)th array reference in the body of loop nest \( L_i \).

While executing, loop nest \( L_i \) is assumed to access \( n \) arrays, \( V_1, V_2, \ldots, V_n \). We use \( V \) to represent a set comprised of these \( n \) arrays. The array element accessed by \( a_j(\vec{I}) \) \((j = 1, 2, \ldots, m)\) can be represented as \( V_i[F(\vec{I})] \) \((i = 1, 2, \ldots, n, j = 1, 2, \ldots, m)\), where \( V_i \) is the name of the array and function \( F \) maps iteration vector \( \vec{I} \) to a vector of subscripts for array \( V_i \). Specifically, \( F(\vec{I}) \), which maps \( k \) loop iterators into \( d \) array indices, where \( k \) is the depth of the loop nest and \( d \) is the dimensionality of the array, can be defined as:

\[
F(\vec{I}) = M\vec{I} + \vec{\sigma},
\]

(4.4)

\footnote{If \( L_i \) is not perfectly nested, one can use techniques such as code sinking \cite{23} to make it perfectly nested.}
where \( M \) is a \( d \times k \) matrix (called the access matrix), \( \vec{I} \) is a \( k \)-element iteration vector, and \( \vec{o} \) is an offset vector [71].

We also assume that the multi-speed disks considered in this work provide \( l \) different rotational speeds: \( RPM = (1, 2, \ldots, l) \), where 1 represents the lowest disk speed and \( l \) corresponds to the highest disk speed available.

Lastly, let us define the disk-layout for each array \( (V_i) \) using a triplet of the form defined in Section 2.4.1. Using the disk layout abstraction, the layout of array \( V_i \) in Figure 4.1(b) can be represented as \( (d1, 4, S) \). The compiler approach described in the next section determines a prefetch distance for each array access in the application code, a rotational speed for each disk in the storage system, and a data layout for each disk-resident array manipulated by the application.

4.4.2 Locality Analysis

To exploit low-speed disks using prefetching in order to save energy, the proposed prefetching algorithm needs to analyze the data locality exhibited by each loop nest \( L_i \) in program \( P \). Given the mathematical representation discussed in Section 4.4.1, temporal reuse is said to occur between two loop iterations \( \vec{I}_1 \) and \( \vec{I}_2 \) whenever \( \vec{F}(\vec{I}_1) - \vec{F}(\vec{I}_2) = \vec{o} \). That is, temporal reuse occurs whenever the difference between the two loop iterations lies in the nullspace of \( M(\vec{r}) = \vec{o} \), i.e., \( \text{span}(M) \). Spatial reuse, on the other hand, is said to occur when two different loop iterations access the same row (in a given array) [71]. To extract the spatial reuse vector space, we simply replace the last row in \( M \) with zeros to create a reduced access matrix, \( M_S \), and solve for nullspace of \( M_S \), which gives us \( \text{span}(M_S) \). After determining the temporal/spatial reuse vector spaces, we next choose the set of innermost loop iterators that can exploit reuse. This is called localized iteration space [71]. This space captures only those loops for which data reuse can result in data locality. In this context, to translate the obtained reuses to locality, we need to take into account the loop iteration count and available memory capacity. Since the loop bounds are assumed to be known at the compile time (if not, we make use of available profile data), one can determine the set of innermost loops whose accessed data fit in the main memory capacity. Data locality is then captured by intersecting the reuse vector space with the localized iteration space, where both are represented
by vector space notation. These steps to analyze reuse and data locality exhibited in the given programs are fundamentally unaltered from those developed in the context of conventional I/O prefetching [18]. However, to support prefetching to multi-speed disks for reducing disk power consumption, we need to be careful in selecting prefetch distance for every disk-resident array references, as will be discussed in detail below.

### 4.4.3 Unified Algorithm

Using the obtained the vector space representation of data locality exhibited by each loop \( L_i \), the proposed approach next determines prefetch distance (\( d \) value in Equation 4.1) for each array reference \( V_i[F(I)] \) made by the loop body of nest \( L_i \). Note that, once the \( d \) value is calculated and reference \( V_i[F(I)] \) is found to have spatial locality on \( i \)th loop, the \( i \)th loop is strip-mined, where \( 1 \leq i \leq k \) and \( k \) is the depth of loop nest. Generally, prefetches are software pipelined around this \( i \)th loop that changes the value of the array-indexing function \( (V_i[F(I)]) \). This chosen loop is called the pipeline loop. As mentioned in the previous section, if we put the data in low-speed disks, the prefetch distance linearly increases with respect to disk I/O time (i.e., the \( T_d \) value in Equation (4.1)), while power consumption is quadratically reduced by the amount of disk speed scaling [22]. Therefore, we need to tune the prefetch distance based on the disk speed, and in fact, the proposed approach determines them together, as explained below.

In the first step of the proposed energy-aware prefetching algorithm, we determine the disk speeds that will provide the maximum energy savings for each array in the application code. To do this, we process array references in the code one by one. In processing an array reference, we consider all possible disk speeds (RPM levels) and select the one that brings the maximum energy savings without performance penalty. It needs to be noted that we may not always select the minimum RPM level for a given array access because there may not be sufficient number of iterations in the loop nest where this array reference appears\(^2\). Therefore, at the end of this first step of the proposed approach, it determines the preferable disk

\(^2\)An alternate approach would be inserting the prefetch call for a given loop nest in one of the preceding loop nests; but, this makes code generation extremely difficult; so, we did not explore this option further.
INPUT:
Input program, \( P = (L_1, L_2, \ldots, L_s); \)
Available disk speeds, \( RPM = (1,2,\ldots,l);\)

OUTPUT:
Determined RPM-group(i), where \( 1 \leq i \leq l;\)

\( T_{pf} = \) the number of cycles for \( PF \) instruction;
for each \( V_k \in V \) // for each array;
\( G[V_k] = \emptyset; \) // possible disk speeds for each array;

// repeat for each loop nest \( L_i. \)
for each \( \mathcal{L}_i \in \mathcal{P} \} \{
\ s_i = \) number of cycles need to execute the loop body of \( L_i; \)
for \( j = 1 \) to \( l \} \) // for each RPM available
  // repeat for all array reference in \( \mathcal{L}_i \)
  // assume that \( a_i(T) \) accesses array element \( V_k[F(T)]; \)
  for each array reference \( a_i(T) \} \{
    \text{calculate I/O latency, } T_d(j); \text{ when } RPM \text{ is } j; \)
    // determine prefetch distance, \( d_j, \) at \( j^{th} \) RPM.
    \( d_j = \lceil \frac{T_d(j)}{s_i + T_{pf}} \rceil; \)
    if \( (d_j > \text{total number of iterations for the pipeline loop}) \)
    \( G[V_k] = G[V_k] \cup \{j\}; \)
  \}
\}

// RPM-group(l) generated by adding maximum value from set \( G[V_i]; \)
for each array \( V_i \} \{
\ l = \{ x | x \in G[V_i] \text{ and } \text{MAX}(G[V_i]) \}; \)
RPM-group(l) = RPM-group(l) \cup \{V_i\};
\}

**Figure 4.4.** Disk speed detection algorithm.

speed for each array reference. However, if a disk-resident array can be accessed from within multiple loop nests, we set the disk speed for that array to the highest speed among all the preferable speeds for all the references to that array. The algorithm that selects the most suitable disk speeds to be used for each array is given in Figure 4.4. The for-each loop in this algorithm goes over the loop nests in the application and the references in them and determines the required disk speed. The for-loop at the end of the algorithm, on the other hand, selects the required RPM level for each array (each \( V_i \)). Note that, at the end of this first step, it also determines the prefetch distances for all array references, in addition to determining the preferable disk speeds for disk-resident arrays, using the approach explained in the first two paragraphs of this section. To summarize, in the first step, we determine both prefetch distances and preferable disk speeds for arrays.

In the next step of the proposed approach, it determines the disk layouts of the arrays in the application. In order to do this, we first form what we call the RPM-
**INPUT:**
- Input program, $P = (L_1, L_2, \ldots, L_s)$;
- Determined RPM-group($i$), where $1 \leq i \leq l$;

**OUTPUT:**
- Determined data layout for each array;

1. $\text{tot\_disks} =$ total number of disks available;
2. $\text{init\_disk} = 0$;
3. $weight[V_i]$: the number of accesses made to $V_i$ within $P$;
4. $weight[V]$: the number of accesses made to all arrays within $P$;

// determine stripe\_factor for $V_i$ with same disk speed
// based on the sum of $weight[V_i]$ in RPM-group($i$).
for $i = 1$ to $l$ {
   // for each RPM-group
   for all $V_i \in$ RPM-group($i$)
      $\text{sum} += weight[V_i]$;
      $\text{stripe\_factor}(V_i) = \text{tot\_disks} \times \lceil \frac{\text{sum}}{weight[V_i]} \rceil$;
      $\text{tot\_disks} -= \text{stripe\_factor}(V_i)$;
}

// determine start\_disk for each array $V_i$
// based on the determined stripe\_factor for each array.
for $i = 1$ to $l$ {
   $\text{start\_disk} (V_i) = \text{init\_disk}$;
   $\text{init\_disk} += \text{stripe\_factor} (V_i)$;
}

**Figure 4.5.** Data layout detection algorithm.

**groups.** An RPM group holds the arrays that require the same RPM level. Each RPM-group is also attached a $weight$, which captures the sum of the number of accesses to the elements of the arrays in that RPM-group. The proposed approach next determines the number of disks that will be assigned to each RPM-group. We currently perform this by distributing the available disks (actually I/O nodes as mentioned in Section 2.2) across the RPM-groups based on their weights in a proportional manner. More specifically, an RPM-group with a larger weight gets assigned more disks than an RPM-group with a lower weight. The reason is that, by assigning more disks to the RPM-group with larger weight, one can exploit the aggregated bandwidth and parallelism presented by multiple disks better. In other words, assigning more disks to the heavy-weighted RPM-group tends to buy more performance benefits. After an RPM-group is assigned its disks, the arrays in that group are striped over those disks using conventional striping. Note that, at the end of this second step of the proposed approach, we fix the disk layout of all disk-resident arrays in the application. The algorithm for determining the disk layouts of arrays is given in Figure 4.5.

The last step of the proposed approach is to restructure the application code
INPUT:
A loop nest \( \mathcal{L} \) for \( \vec{I} \in [\vec{L}, \vec{U}], step \vec{b} \langle a_1(\vec{I}), \ldots, a_m(\vec{I}) \rangle \)
\[ \vec{L} = (l_1, l_2, \ldots, l_n)^T \]
\[ \vec{U} = (u_1, u_2, \ldots, u_n)^T \]

OUTPUT:
Transmformed loop nest \( \mathcal{L}' \); for \( \vec{I}' \in [\vec{L}', \vec{U'}], \langle a_1(\vec{I}'), \ldots, a_m(\vec{I}') \rangle \)

// assume that \( \vec{I}_p \in (I_1, I_2, \ldots, I_k)^T \) is the selected pipeline loop for each \( I_p \) selected for \( V_i \) 
add a new controlling loop denoted by \( \mathcal{H}_p \) \( \langle u_p', l_p' \rangle \) to the loop nest \( \vec{I} \)
\ such that \( \vec{I}' = (I_1, \ldots, I_p, \ldots, I_k)^T \);
// calculate new loop bounds for \( \mathcal{H}_p \) and \( I_p \).
\( [l_p', u_p'] = [l_p, u_p] \);
\( b_p' = \) loop step needed to strip-mine \( I_p \) loop;
add \( b_p' \) into the loop step vector, \( \vec{b} \)
\ such that \( \vec{b}' = (b_1, \ldots, b_p', b_p, \ldots, b_n) \);
\( [l_p, u_p] = [l_p', l_p'+b_p'] \);
emit “for \( \vec{I}' \in [\vec{L}', \vec{U'}], step \vec{b}' \langle \); // insert prefetch instruction.
for all array references being prefetched
emit “PF(\( V_i [\vec{F} [\vec{I}']])”;
// copy loop body from original loop body.
emit “a_1(\vec{I}'), \ldots, a_m(\vec{I}')”;
emit “\);”;

Figure 4.6. Code restructuring algorithm.

Figure 4.7. The three steps of the proposed approach to energy-aware data prefetching.

to insert prefetch instructions. Since the prefetch distances for all array references have already been determined by the first step explained above, the third step uses this information and restructures the application code accordingly based on the strip-mining based approach proposed by Brown et al [18]. Figure 4.6 shows the pseudo-code for the algorithm that modifies the application code. The overall view of the proposed approach to energy-aware prefetching is depicted in Figure 4.7.

As explained above, the proposed approach determines prefetch distances, data layouts and disk speeds in a unified setting. However, it can also be modified to
work with given data layouts and disk speeds. If the data (array) layout and disk speeds are fixed, the proposed hoisting algorithm determines prefetch distance based on existing information and then modifies the code accordingly. As an example for this case, let us consider the code fragment shown in Figure 4.8(a). As shown in Figure 4.8(b), arrays \( V_1 \) and \( V_2 \) are striped across three disks, each of which has a different rotational speed. In this case, we split the original loop nest into a series of smaller loop nests such that each split loop nest accesses the data stored in the disk with a particular speed. We use the Omega library [72, 57], a polyhedral tool, to generate these restructured loop nests using the given data layouts and the data access patterns extracted by the compiler. In this example, we see that, we can divide the original loop nest into three loop nests. We then calculate the prefetch distance for the reference in each loop nest, \( d_i \) (1 \( \leq i \leq 3 \)) and restructure the nests accordingly. The transformed code is illustrated in Figure 4.8(c). This small example shows that the proposed approach can be applicable even if the data layouts and the disk speeds are determined a priori.

\[ L: \text{for } i = 0 \text{ to } N - 1 \{ \]
\[ \ldots V_1[i] \ldots \]
\[ \ldots V_2[i] \ldots \]
\[ \} \]

(a) Original code fragment.

\[ L_1: \text{for } ii = 0 \text{ to } \frac{(N-1)}{3} \text{ to } \frac{2(N-1)}{3} - d_1, \text{ step } b_1 \{ \]
\[ \text{PF } (&V_1[i + d_1]); \text{ PF } (&V_2[i + d_1]); \]
\[ \text{for } i = ii \text{ to } ii + b_1 \{ \]
\[ \ldots V_1[i], V_2[i], \ldots \]
\[ \} \]

(b) Data layouts and disk speeds.

\[ L_2: \text{for } ii = \frac{(N-1)}{3} \text{ to } \frac{2(N-1)}{3} - d_2, \text{ step } b_2 \{ \]
\[ \text{PF } (&V_1[i + d_2]); \text{ PF } (&V_2[i + d_2]); \]
\[ \text{for } i = ii \text{ to } ii + b_2 \{ \]
\[ \ldots V_1[i], V_2[i], \ldots \]
\[ \} \]

(c) Transformed code fragment. The three loops show only the steady-state of the pipelined loops.
\[ \mathcal{L}_1: \text{for } i = 1 \text{ to } N - 2 \{
\text{for } j = 1 \text{ to } N - 2 \{
\ldots V_1[i][j] \ldots
\ldots V_2[i][j] \ldots
\ldots V_3[i][j] \ldots
\}\}
\]
\[ \mathcal{L}_2: \text{for } j = 0 \text{ to } N - 1 \{
\ldots V_1[N][j] \ldots
\}\}
\[ \mathcal{L}_3: \text{for } i = 0 \text{ to } N - 1 \{
\ldots V_2[i][N] \ldots
\}\}
\]

(a) Original loop nests.

<table>
<thead>
<tr>
<th>RPM-group</th>
<th>arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>15K</td>
<td>(V_2)</td>
</tr>
<tr>
<td>12K</td>
<td>(V_1)</td>
</tr>
<tr>
<td>6K</td>
<td>(V_3)</td>
</tr>
</tbody>
</table>

(c) Determined RPM-groups.

\[ \mathcal{L}'_1: \text{for } ii = 1, \ldots, N - 4, \text{ step } b_1 \{
\begin{align*}
\text{PF } (\&V_3[ii+2][j]) &; \\
\text{for } jj = 1, \ldots, N - 2 - (N/4), \text{ step } b_2 \{
\text{PF } (\&V_1[ii][jj+(N/4)]) &; \\
\text{for } i = ii \text{ to } ii+b_1 \{
\text{for } j = jj \text{ to } jj+b_2 \{
\ldots V_1[i][j] \ldots
\ldots V_2[i][j] \ldots
\ldots V_3[i][j] \ldots
\}\}
\}
\}
\]
\[ \mathcal{L}'_2: \text{for } jj = 0, \ldots, (N-1) - (N/4), \text{ step } b_3 \{
\begin{align*}
\text{PF } (\&V_1[N][jj+(N/4)]) &; \\
\text{for } j = jj \text{ to } jj+b_3 \{
\ldots V_1[N][j] \ldots
\}\}
\}
\]
\[ \mathcal{L}_3: \text{for } i = 0 \text{ to } N - 1 \{
\ldots V_2[i][N] \ldots
\}\}
\]

(e) Transformed loop nests. Both \(\mathcal{L}'_1\) and \(\mathcal{L}'_2\) show only the steady state of the pipelined loops.

(b) Locality analysis for each array reference.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Locality</th>
</tr>
</thead>
<tbody>
<tr>
<td>(V_1[i][j]), (V_2[i][j]), (V_3[i][j])</td>
<td>(i) = none, (j) = spatial</td>
</tr>
<tr>
<td>(V_1[N][j])</td>
<td>(j) = spatial</td>
</tr>
<tr>
<td>(V_2[i][N])</td>
<td>(i) = none</td>
</tr>
</tbody>
</table>

(d) Determined disk speeds and data layouts for arrays.

Figure 4.9. An example application of our hoisting algorithm.

### 4.4.4 Example

We now give a more detailed example to show how our algorithm described in Section 4.4 works in practice. The original code fragment shown in Figure 4.9(a) has three loop nests, \(\mathcal{L}_1\), \(\mathcal{L}_2\), and \(\mathcal{L}_3\) and it manipulates three different disk-resident arrays, namely \(V_1\), \(V_2\), and \(V_3\), using different indexing functions in each loop nest.
For illustrative purposes, let us assume that all the arrays are of the same size, $N \times N$. Let us further assume that we have four possible RPM levels, namely, 15K, 12K, 9K, and 6K RPMs, for each disk in the system. Originally, all disks are assumed to be run at 15K RPM. Based on the locality analysis, we can obtain the temporal/spatial locality information of $P$, as shown in Figure 4.9(b). This locality information indicates that, in the first loop nest ($L_1$), all three array references have spatial locality in the $j$ loop. Since $j$ is chosen as the pipeline loop, we subsequently calculate the prefetch distance ($d_i$) for every possible disk speeds, i.e., 15K, 12K, 9K, and 6K (using the algorithm in Figure 4.4). For ease of illustration, let us assume that the determined $d_i$ values for disk speeds 15K, 12K, 9K, and 6K are $(N/8)j$, $(N/4)j$, $Nj$, and $2Nj$ loop iterations, respectively. This implies that, when considering $L_1$ alone, we can store all arrays ($V_1$, $V_2$, and $V_3$) in 6K RPM disks, which require $2Nj$ loop iterations to schedule-ahead the disk access since the latency incurred by 6K RPM disks can be eliminated by choosing next surrounding loop nest, i.e., $i$ loop as the pipeline loop. However, since both arrays $V_1$ and $V_2$ are accessed again in nests $L_2$ and $L_3$ respectively, possible disk speeds for $V_1$ and $V_2$ arrays are also dependent on the $L_2$ and $L_3$ nests. After processing all the three loop nests, we obtain the possible RPMs for each array, i.e., we have $G[V_1] = \{6K, 12K\}$, $G[V_2] = \{6K, 15K\}$, and $G[V_3] = \{6K\}$. The RPM-groups can be obtained by aggregating the maximum possible RPM from each $G[V_i]$, and they are listed in Figure 4.9(c). This indicates that, for the array $V_1$ accessed by both $L_1$ and $L_2$, we can assign 12K RPM because the $j$ loop in $L_2$ is sufficient for hiding latency with a $(N/4)j$ prefetch distance. The disk speed originally assigned to $V_2$ remains in 15K RPM since the obtained reuse vector for array reference in $L_3$ indicates that there is no inherent spatial locality. Based on these disk speeds determined, the resulting data layouts (determined using the algorithm in Figure 4.5) and the transformed code fragment (obtained using the algorithm in Figure 4.6) are given in Figures 4.9(d) and (e), respectively. In this example, since all three arrays are of the same size, we assign two disks (out of six disks) per each array (and per RPM-group in this case), and the speed of each disk is set to the RPM level as determined by our algorithm. As we can see from Figure 4.9(d), we can save disk energy consumption by running four disks (out of a total of six disks) at lower speeds. Also important to note that the performance of this transformed
code is not expected to be any worse than an alternate code that uses compiler-based I/O prefetching, such as [18], that does not care about energy consumption.

4.5 Experimental Evaluation

4.5.1 Simulation Platform

To evaluate the effectiveness of our approach in reducing disk energy consumption, we implemented a simulation platform using DiskSim [41]. We assumed that each I/O node has one disk; that is, no further striping is applied within any I/O node. DiskSim is driven by externally-provided disk I/O traces, which are generated by our trace generator. The trace generator generates disk I/O traces, extracted from the disk layout information and the disk access pattern, the latter of which can be obtained either through profiling or static analysis. We modeled an IBM Ultrastar 36Z15 disk [2] and its relevant power and performance characteristics are shown in Table 4.1. Since we use multi-speed disks running at different RPM levels, we model the performance and energy values at every possible disk speed used. Based on the data from a conventional IBM36Z15 disk, whose rotational speed is 15K RPM, we obtained the performance and energy consumption values at idle and active state by using the quadratic disk power model described in [22]. The energy and performance values for these multi-speed disks are also given in Table 4.1.

For each application in our experimental suite, we performed experiments with three different schemes:

Table 4.1. Major simulation parameters and their default values.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk Parameters</td>
<td></td>
</tr>
<tr>
<td>Disk Model</td>
<td>IBM Ultrastar 36Z15</td>
</tr>
<tr>
<td>Interface</td>
<td>SCSI</td>
</tr>
<tr>
<td>Storage Capacity</td>
<td>18.4 GB</td>
</tr>
<tr>
<td>Disk Cache Size</td>
<td>4 MB</td>
</tr>
<tr>
<td>Internal Transfer Rate</td>
<td>55 MB/sec</td>
</tr>
<tr>
<td>Disk Performance and Energy Model</td>
<td></td>
</tr>
<tr>
<td>Rotation speed</td>
<td>15K/12K/9K/6K RPM</td>
</tr>
<tr>
<td>Average rotational latency</td>
<td>2.0/2.5/3.33/5.0 ms</td>
</tr>
<tr>
<td>Power (active)</td>
<td>13.5/11.3/9.1/6.9 W</td>
</tr>
<tr>
<td>Power (idle)</td>
<td>10.2/8.66/7.12/5.58 W</td>
</tr>
<tr>
<td>Striping Information</td>
<td></td>
</tr>
<tr>
<td>Stripe size</td>
<td>64 KB</td>
</tr>
<tr>
<td>Stripe factor (number of disks)</td>
<td>16</td>
</tr>
<tr>
<td>Starting iodevice (starting disk)</td>
<td>1 (the first disk)</td>
</tr>
</tbody>
</table>
• **Base**: This is the baseline version that does not employ any prefetching scheme. It executes benchmark programs on a disk subsystem where all disks run at the highest available speed, i.e., 15K RPM. All the reported disk energy and performance numbers presented later in this section are given as normalized values with respect to the corresponding numbers obtained using this version (which are given in the last two columns of Table 4.2).

• **PF**: This scheme corresponds to the conventional I/O prefetching approach, as explained in [18]. The underlying disk speed is fixed at the default RPM level (15K) and the data layouts are exposed to the compiler. As in the base version, we striped all arrays across all disks in the system. Given the disk speeds and data layout of arrays, this scheme restructures the loop nests in the application code to hide I/O latency incurred by accessing high-speed disks.

• **PF+**: This scheme corresponds to our energy-aware data prefetching approach, as has been discussed in detail in Section 4.4. As discussed earlier, it determines the disk speeds for all disks in the system and the data layout for each disk-resident array. Based on these determined parameters, it also restructures loop nests.

In our experiments, we used four SPEC2000 float-point benchmark programs [42]. The important characteristics of these benchmark programs are given in Table 4.2. We made the array data manipulated by these benchmark programs disk-resident; so, accessing an array data during execution results in a disk I/O of a block size (default block size is 8KB), unless the access is captured in the cache. To be fair in evaluating our approach, however, we also optimized these benchmark codes (even the base version) so that the number and volume of I/O accesses are minimized as much as possible. That is, our benchmarks are highly optimized as far as their I/O behavior is concerned. Also, to complete our simulations within a reasonable amount of time, we focused only on the loop nests whose cumulative I/O times account for more than 90% of the total I/O time of each benchmark. Using the default simulation parameters given in Table 4.1, the baseline energy and performance results are given in the last two columns of Table 4.2. These baseline results are obtained by executing our benchmark programs on a disk subsystem.
Table 4.2. Benchmarks and their characteristics.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Base</th>
<th>Exec Size (GB)</th>
<th>Energy (J)</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>171.swim</td>
<td>Shallow Water Modeling</td>
<td>115.2</td>
<td>56648.1</td>
<td>301.9</td>
<td></td>
</tr>
<tr>
<td>172.mgrid</td>
<td>Multi-grid Solver: 3D Potential Field</td>
<td>95.5</td>
<td>175470.3</td>
<td>1066.1</td>
<td></td>
</tr>
<tr>
<td>173.aplu</td>
<td>Parabolic/Elliptic Partial Differential Equations</td>
<td>99.2</td>
<td>121798.5</td>
<td>736.9</td>
<td></td>
</tr>
<tr>
<td>301.apsi</td>
<td>Meteorology: Pollutant Distribution</td>
<td>107.9</td>
<td>456479.1</td>
<td>2786.7</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.10. Normalized energy consumptions.

Figure 4.11. Normalized execution times.

where all disks run at the highest RPM level (15K). As mentioned earlier, the results which will be given in the next subsection, are normalized with respect to the values in these last two columns.

4.5.2 Results

The bar-chart shown in Figure 4.10 gives the normalized energy consumptions of the benchmark programs in our experimental suite. One can make several observations from these results. First, PF brings an average disk energy savings of 39.6% across all four benchmarks compared to the Base version. These savings are due to the reductions in disk idle times. The second observation one can make from this bar-char is that the PF+ version (our approach) achieves additional energy savings, 19.6% on average when all benchmarks are considered. This indicates that our approach successfully determines the lowest possible rotational speed for each disk and the corresponding disk layouts. As opposed to the PF version, our approach is able to reduce the energy spent in active periods.

We now present the performance results obtained. The normalized execution times for our benchmarks are presented in Figure 4.11. One can see from this graph
that the PF scheme reduces execution time by 41.3% compared to the Base scheme. This result shows that prefetching is beneficial in enhancing performance by hiding the latency incurred by I/O requests. One can also see that the performance of the PF+ scheme is almost same as that of the PF scheme (the execution time difference between PF and PF+ is negligible, i.e., below 1%). This suggests that our approach can achieve a significant amount of disk energy savings with little impact on the performance improvement achieved by the PF scheme.

4.5.3 Sensitivity Analysis

In our next set of experiments, we perform a sensitivity analysis, varying simulation parameters pertinent to disk striping. Specifically, we vary the stripe size and stripe factor (the number of disks used for striping) to see how our approach gets affected. For illustrative purposes, we choose one benchmark, 173.applu, and conduct all sensitivity analysis using that benchmark. However, the results we observed extend to other three benchmarks as well. Figure 4.12 gives the normalized energy consumptions with the different stripe sizes (ranging from 32KB to 256KB). Recall from Table 4.1 that the default stripe size was 64KB. The values of the all other simulation parameters are fixed at the values given in Table 4.1. We see from these results that the energy savings achieved by our scheme are slightly increasing as we increase the stripe size. This can be explained as follows. When the stripe size increases, a given disk tends to service I/O requests for a longer period of time. This in turn leads to fewer disks being involved in processing the I/O requests, thereby increasing the idle periods of other disks. Consequently, these longer idle periods contribute to reduction in disk energy consumption.

In our next sensitivity experiment, we measured the impact of the different stripe factors (i.e., the total number of disks used for striping). Figure 4.13 gives the normalized energy consumptions with the different stripe factors (ranging from 8 to 64 disks). We observe from these results that the energy savings our approach achieve are slightly decreasing as the number of disks increases. This is because, as we increase the number of disks used in striping, this increases the overall idleness of disks. And, since a disk in the idle state consumes almost same amount of energy as it would consume in the active state, this in turn increases the overall
energy consumption. Still, the experimental results given in Figures 4.12 and 4.13 clearly show that our approach is successful across a range of values for stripe sizes and the number of disks.

4.6 Summary

I/O prefetching has been widely used in many I/O-intensive applications to improve I/O performance by hiding the latency. Unlike conventional performance-oriented I/O prefetching, this section presents a compiler-directed energy-aware prefetching scheme for disk-intensive scientific applications. The proposed approach determines, in a unified setting, the prefetch distances for disk access (I/O) instructions, the disk speeds for all disks in the storage system, and the data (array) layouts on the disks, given an application program.
Chapter 5

Runtime Support for Multiple Applications

5.1 Introduction

Reducing power consumption of disk systems can be very important for high-performance architectures where disks are responsible for a large fraction of overall power budget [10, 73, 15, 74]. As system sizes and capabilities approach the petascale range, one can expect scientific applications to be even more data hungry in the future, which means more frequent use of the disk storage systems. Prior research studied disk power reduction techniques from both the hardware and software perspectives. Hardware related work in this area includes spinning down idle disks [19, 20] and employing multi-speed disks [10, 22]. Software work on the other hand focuses on file system based approaches such as [54] and compiler related studies [47, 75, 76].

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

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

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

We implemented our runtime system support within PVFS2, a parallel file system [26] and performed experiments with four scientific applications that process disk-resident data sets. Our experimental results are very promising: We achieve, under the default values of our simulation parameters, 39.9% and 19.4% savings in disk power consumption, over the pure hardware based and the pure compiler based schemes, respectively. We further show in this chapter that our scheme can achieve consistent energy savings with a varying number and mix of applications and different disk layouts of data.
/* Loop on horizontal slices */
for ih=0, R, 1 {
    /* Read next block of matrix A */
    MPI_File_read_at_all (fh_A, ···);
    /* Loop on vertical slices */
    for iv=1, R, 1 {
        /* Read next block of matrix B */
        MPI_File_read_at_all (fh_B, ···);
        /* Compute block product */
        for i=0, Nb, 1
            for j=0, Nb, 1
                for k=0, Nb, 1
                    C[i][j] += A[i][k] * B[k][j];
        /* Write block of C */
        MPI_File_write_at_all (fh_C, ···);
    }
}

/* Set the disk speed to 6K RPM */
set_speed (011100, 6000, 0);

/* Compute block product */
for i=0, 3Nb/4, 1
for j=0, Nb, 1
    for k=0, Nb, 1
        C[i][j] += A[i][k] * B[k][j];
/* Set the disk speed to 15K RPM */
set_speed (011110, 15000, 1);

/* Write block of C */
MPI_File_write_at_all (fh_C, ···);
}

(a) Original code fragment.

Figure 5.1. Out-of-core matrix-matrix multiplication loop nest written using the MPI IO calls. Each data slice (tile) is of size \( N_b \times N_b \) array elements, and all three matrices, A, B, and C, have \( R \times R \) number of slices, which corresponds to the number of processors.

5.2 Compiler Support

Our focus is on MPI-IO based parallel scientific applications. MPI-IO, which is the I/O part of the MPI-2 standard, is an interface that supports parallel I/O operations and optimizations [77, 78]. Our target applications access file-based data frequently for different purposes, e.g., reading and writing large data sets for out-of-core computations, storing the results of intermediate computations for long-running applications, and reading back data stored for checkpointing or visualization. One key characteristic of these applications is that, through automated code analysis, an optimizing compiler can identify logical file addresses along with the request size of file blocks. To explain this better, let us consider the out-of-core blocked matrix multiplication nest shown in Figure 5.1(a). In this example, each processor accesses all three disk-resident arrays, A, B, and C, and opens sub-files that correspond to one row-block from A, and one column-block from B, and their intersection from C. It then reads data from A and B, carries out the corresponding
partial multiplication in memory, and stores the results in C. Note that, the code given in Figure 5.1(a) will be executed using multiple processors and each processor performs both the I/O and computation assigned to it. As we can see from this code, it involves a substantial amount of disk I/O followed by some computation associated with it. Consequently, one can conceivably implement proactive disk power management by inserting explicit disk power management calls before and after the I/O phases. Since many scientific applications exhibit this type of behavior, we use this code fragment, as our example, to describe how a compiler can predict disk access patterns and modify the source code to insert explicit power management calls.

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

We next discuss how the disk layout abstraction discussed in Section 2.2 can be used to determine the disk being accessed by each I/O phase. As mentioned earlier, file striping is a technique that divides a large chunk of data into small data blocks (stripe) and stores these blocks on separate disks in a round-robin fashion (as depicted in Figure 2.1). This permits multiple processes to access different portions of the data concurrently without much disk contention. In addition to providing file striping, many parallel file systems today also provide a hint mechanism through which one can specify user-defined or manual file striping, as will be explained
below. In this chapter, we represent the disk layout of a file using the following triplet:

\[(\text{Base, Striping Factor, Striping Unit}),\]

where Base is the first disk from which the file gets striped, Striping Factor is the number of disks used for striping, and Striping Unit (or stripe size) is the length of each stripe. In the PVFS2 file system [26], one can change the default striping parameters using the following MPI-IO hints that are supported: \texttt{CREATE\_SET\_DATAFILE\_NODES} (enumeration of disk ID being used for striping), \texttt{Striping Factor}, and \texttt{Stripe Size}. Then, the striping information defined by the \texttt{MPI\_Info} structure is passed to to \texttt{MPI\_File\_open()} calls. When creating a file within the application, this MPI hint information can be made available to the compiler as well. As explained above, the compiler uses this information in conjunction with file offset and request block size to determine the disk access pattern. If the file has already been created on the disk system, we can also obtain its layout by querying \texttt{MPI\_Info} associated with the file in question. The obtained \texttt{MPI\_Info}, which indicates the disk layout of the file, can then be passed to the compiler.

To demonstrate how to extract the disk access pattern using the code fragment shown in Figure 5.1(a), let us assume the disk layout in Figure 5.1(b). Assuming the same stripe size of 64Kbytes for all file stripes, the disk layouts for arrays A, B, C in this example are \((1, 3, 64)\), \((1, 3, 64)\), and \((2, 3, 64)\), respectively. Let us further assume, for illustrative purposes, that the request size of each \texttt{MPI\_Read}/\texttt{MPI\_Write} call is of three stripe units, i.e., \(64\times 3\) Kbytes. Using the data access patterns and the file layout associated with them, the compiler can infer that, four disks \((d_1, d_2, d_3, \text{ and } d_4)\) must be run at the highest disk speed available in order not to degrade the I/O response time. On the other hand, since \(d_0\) and \(d_5\) are determined not to be used while executing the code fragment given in Figure 5.1(a), we can use for them the lowest disk speed available. Since the determined disk access patterns capture both the idle and active periods for each disk and their durations, we can insert (in the code) explicit power management calls. We also want to mention at this point that, in our approach, a power management call inserted by the compiler is actually a \textit{hint} to the runtime system. That is,
the runtime system does not need to obey all the speed-setting calls passed to it. Rather, it uses them to make a globally acceptable decisions regarding disk speeds. The details of how our runtime system makes that decision based on the compiler-inserted calls will be explained in Section 5.3.

As mentioned above, the last part of our compiler support is to insert power management calls (set_speed() calls in our case) in the code. As an example, let us consider the code fragment in Figure 5.1(a) once again. Let us assume, for the sake of illustration, that a disk can spin at four different speeds, namely 6000, 9000, 12000, and 15000 RPM. In the previous step, we determine that, after reading the block of arrays A and B, the speeds of the disks just accessed (\(d_1, d_2, \text{and } d_3\) in this example) can be set to the lowest speed available (6000 RPM). After the computation phase, array C must be written back to the disks. Consequently, the timing for restoring the disk speed to its maximum, 15000 RPM, is very important in order not to incur energy or performance overheads. More specifically, if the speed-setting calls are issued too early, the disk will be placed into the highest speed earlier than necessary. On the other hand, if they are issued too late, this can degrade performance. Since most scientific applications are built using nested loops, we use the number of loop iterations as our unit for deciding the point to insert the explicit disk speed-setting call, which is calculated as:

\[
\left\lceil \frac{T_s}{T_b} \right\rceil,
\]

(5.1)

where \(T_s\) is the cycles required to set the disk speed given as a parameter to set_speed() call, and \(T_b\) is the shortest possible execution latency (in cycles) through the loop body. Once the insertion point for the disk speed-setting call is determined, we use loop splitting [23] to make explicit the point where the call is to be inserted. The format of our disk speed-setting call is as follows:

\texttt{set\_speed} (\(T, S_i, F\)),

where \(T\) is a tag that consists of \(D\) bits, where \(D\) is the number of disks in disk system. The bit in the \(j\)th position of \(T\) (\(0 \leq j \leq D - 1\)) is set to 1 if the MPI-IO call accesses the \(j\)th disk. \(S_i\) (\(1 \leq i \leq m\)) is the \(i\)th speed level, and the last argument to the set\_speed() call, \(F\), is a flag which is set to 1 when the requested speed setting is for the upcoming I/O phase. Otherwise, \(F\) is set to 0 and this
means that the disk will not be used for a certain period of time. Going back to our example code fragment in Figure 5.1(a), let us assume that the number of loop iterations required to change the disk speed from 6000 RPM to 15000 RPM is one fourth of the total number of iterations of the outermost loop, \( i \). Therefore, the compiler splits the original loop nest into two parts: one with the first \( 3/4 \)rd of iterations and one with the remaining \( 1/4 \)th of iterations, and inserts the \texttt{set\_speed}(011110, 15000, 1) call between these two newly-generated loop nests, as shown in Figure 5.1(c).

Our compiler algorithm for determining disk access patterns and identifying the desirable disk speed for each I/O call is given in Figure 5.2. This algorithm takes an input program along with available disk speeds and inserts explicit \texttt{set\_speed} calls (hints) into the program. The first step of our compiler algorithm is to determine the disk access patterns for each I/O call in the input program. As a result of the first step, the compiler determines the \( T \) parameters, which capture disk access patterns for each I/O call. The second step of our algorithm involves analyzing the loop nest that contains I/O calls and finding the desirable disk speed for each disk accessed by each I/O call. Based on the determined disk speed, the compiler then identifies the splitting point in the original loop nest. The last step of the algorithm is to transform the loop and insert the \texttt{set\_speed() calls} in the determined points in the code.

Before moving to the next section, we would like to mention that the decisions made by the compiler explained so far are only valid from the perspective of a single application. Consequently, they may not be suitable when other (concurrently executing) applications that use the same set of disks are considered. Our proposed runtime support, which is explained next, is designed to address this problem. We also need to mention however that the compiler-determined preferable speeds are still very important. This is because if the optimized application happens to be the sole client of a particular disk, the runtime system will just use that preferable speed (indicated by the compiler) for the disk. Even if this application is not the only client of a disk, the runtime system still uses this hint in deciding the disk speed.
**INPUT:**
Input program, \( P \);
Available disk speeds, \( DS = (S_1, S_2, \ldots, S_m) \);

**OUTPUT:**
Transformed program, \( P' \);

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

/* Step 1: Determine disk access patterns */
for each I/O call \( \in P \) { 
  determine the I/O call parameters, i.e., file id, file offset, and size;
  query \( DL \) of file id;
  if (size \( \leq \text{striping unit} \) ) { /* small request size */
    \( i = (\text{base} + (\text{file offset} \% \text{striping unit}) \% D \); 
    set \( i \)th bit to 1 in \( T \);
  }
  else { /* large request size */
    for (\( j = \text{base}; j < \text{base}+\text{striping factor}; j=j+1 \))
      set \( j \)th bit to 1 in \( T \);
  }
}

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

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

**Figure 5.2.** Our compiler algorithm, in pseudo code, that determines the disk access patterns and finds the most desirable disk speeds for the disks accessed by each MPI-IO call.

### 5.3 Runtime System Support

We first address the problem of executing multiple applications concurrently, each of which is transformed to insert explicit disk speed-setting calls. Figure 5.3 shows the sketches of two applications, A1 and A2, that are modified by the compiler (as explained in Section 5.2). In this execution scenario, it is assumed that A1
is determined to have more idle periods than A2. Hence, the speed of the disks accessed by A1 is set to 6000 RPM by the compiler, whereas that of the disks accessed by A2 is set to 9000 RPM. Before the I/O phase of each application starts, the disks are set to maximum speed, 15000. Note that, the second set_speed() call of each application should be issued early enough to hide the time required to restore the speed of disks to the maximum. Figures 5.4(a) and (b) depict the disk power state diagram for A1 and A2, respectively, when A1 is started to execute followed by A2. Both A1 and A2 issue three set_speed() calls (hints) during their execution periods. While two of these calls do not have any impact on A2’s performance and energy behavior, the second call issued by A1 has a substantial impact on the first I/O phase of A2, denoted using IO2 in Figure 5.4(b). This is because the decision on disk speed made at the time when A1 issues set_speed (1110, 6000, 0) is lower than the one required for A2, which is 15000 RPM. In order not to incur a negative impact on the performance of A2, we need to set the speed of the shared disk to 15000 RPM, the higher of the conflicting speeds.

Figure 5.5 illustrates how the proposed runtime system support handles the software directed disk power management calls inserted by the compiler. To make a decision that would be agreeable to all applications, the runtime system needs to intercept each disk power management call inserted by the compiler, and it should maintain the current speed and the requested one for every disk in the system. For this purpose, it maintains an array of disk speeds, each of which corresponds
to the current speed of each disk. For example, assuming we have four disks in the system, \{15K,15K,12K,9K\} means that the current speed of \(d_0\) and \(d_1\) is 15K, and the current speeds of \(d_2\) and \(d_3\) are 12K and 9K, respectively. The decision for selecting the most appropriate speed for each disk depends primarily on the current disk speed and the requested disk speed, denoted as \(S_C\) and \(S_R\) respectively in Figure 5.5. If the current speed of a disk used by an application is \(S_C\) and another application issues a \texttt{set-speed()} call requiring speed \(S_R\) for the same disk, this request will be granted if and only if \(S_C < S_R\). In other words, an application is allowed to speed up a disk but it is not allowed to slow down a disk if that disk is also being used at the same time by another application. To keep track whether a disk is shared or not, we attach a counter to each element of the disk speed array, explained above. The counter attached to a disk is increased if and only if a \texttt{set-speed()} call issued by an application and the \(F\) parameter to it is set to 1. Recall that, the \(F\) parameter in the \texttt{set-speed()} call is set to 1 when the compiler predicts that the disk should be spinning at the specified speed in order not to incur performance degradation. The counter is decreased when an application releases its use of disks by issuing the \texttt{set-speed()} call with the \(F\) parameter set to 0, or when the application terminates. If the disk is used by only one application (i.e., the counter attached to it is 1), a request to slow down the disk is granted by the runtime system. On the other hand, if the requested speed is equal to the current one, such a request is discarded.

Figure 5.6 shows an example of how our runtime system selects preferable
Figure 5.6. An example that illustrates the selection of preferable disk speeds for a scenario that involves two concurrently-running applications. The disk layouts of the file accessed by A1 and A2 are (0, 3, 64) and (1, 3, 64), respectively. Each entry in the current disk speed array corresponds to the speed of each disk. The number next to each disk speed is the counter value associated with each disk, which indicates how many applications are currently using that disk.

disk speeds based on the hints issued by two applications, A1 and A2, running in parallel. The disk layouts of the files accessed by A1 and A2 are assumed to be (0, 3, 64) and (1, 3, 64), respectively, and it is assumed that we have four disks in the system. Initially, all the entries in the array of current disk speeds are set to zero, as depicted in Figure 5.6. The counter associated with each entry is also initialized to zero. In this example, each application sends three speed-setting calls (denoted as \( S_i \) in the figure) to the runtime system during execution. When our runtime system receives \( S_1 \), it updates the array of current disk speeds accordingly. Since the disk layout of the file accessed by A1 is (0, 3, 64), it sets the speed of the first three disks (i.e., \( d_0, d_1, \) and \( d_2 \)) to 15K. When the runtime system receives \( S_2 \) from A2, it tries to set the speed of the disks accessed by A2, which are \( d_1, d_2, \) and \( d_3 \). The runtime system sets the speed of \( d_3 \) only, and the requests for the disks \( d_1 \) and \( d_2 \) are discarded because the requested speed of \( S_2 \) is the same as the current speed, which is 15K. The third and fourth requests need to be processed carefully because they demand slower speeds than the current one. In processing \( S_3 \), the runtime system first checks whether the disks accessed by A1 are shared or not. Since at this point only \( d_1 \) and \( d_2 \) are shared with A2 (i.e., the counters associated with \( d_1 \) and \( d_2 \) are not 1), the runtime system sets the speed of \( d_0 \) to the requested one, which is 6K (since this disk is not currently shared). After processing \( S_3 \), the corresponding counter is decreased accordingly. Similarly, when processing \( S_4 \), the runtime system allows slowing down the disks accessed by A2 to 9K because both A1 and A2 inform that the disks accessed by them are not used for a certain period.
of time. The remaining two requests, $S_5$ and $S_6$, on the other hand, are granted immediately because they require a higher disk speed than the current one.

### 5.3.1 Implementation

We implemented a prototype of the runtime system within PVFS2 [26], an open source parallel file system designed to run on Linux clusters. We modified three major parts of PVFS2 and MPICH2 [79] to support the proposed runtime system approach. First, as the application code transformed by the compiler contains an explicit power management call, `set_speed()`, and this call is later used by the runtime system layer, we added this interface to ROMIO, which is the part of MPICH2 that contains the implementation of MPI-IO call interfaces to PVFS2. Because any number of executables compiled using the modified MPICH2 can make a `set_speed()` request to the runtime system simultaneously, we have to ensure that processing such requests is performed in an atomic fashion. To achieve this, we used the mutex lock mechanism through the internal lock functions supported by PVFS2.

Second, the array of the current disk speeds is implemented as an internal data structure of the PVFS2 server. Since every PVFS2 server running on each node should know the current status of the disk speed to make their own decisions, we added this data structure to the metadata server so that all PVFS2 I/O nodes can lookup the shared data structure when they make the decision. Specifically, when a `set_speed()` call is received, the PVFS2 server checks to see if the specified disk speed can be set based on the decision made by the runtime module described in Figure 5.5. If the requested speed is granted by the runtime module, it is queued to the job threads of the PVFS2 server.

The last part of our modification to PVFS2 is to emulate multi-speed disks. Because we do not have access to a multi-speed disk system, we need to emulate its behavior to evaluate the proposed approach. There are two unique disk activities of multi-speed disks we have to emulate to obtain correct timing regarding disk accesses. The first activity is changing the speed of the disks, as triggered by the runtime system. The second one is handling disk I/O when the speed is set to a lower one than the maximum speed. To mimic the behavior of these activities, we
used `nanosleep()`, a high resolution sleep function available under Linux/Unix. Specifically, whenever the runtime system makes a decision to change the speed of a particular PVFS2 I/O node (disk), we made that node sleep until the time specified by the argument to the nanosleep call. As explained in [22], the amount of time required for changing the speed of a disk depends on the difference between the current speed and the requested speed, and in our implementation we applied the technique in [22] to calculate this cost. Similarly, the slowed I/O time due to the lower disk speed, which is set by the runtime system, is also imitated by intercepting every I/O thread\(^1\) and involving the required sleep calls before the thread terminates.

5.3.1.1 Handling Hints During Disk Speed Transition

Since each application can issue its own set_speed hints to the disk subsystem, the runtime system can receive such hints while processing prior set_speed hints. More specifically, when the runtime system receives a set_speed hint whose target speed is higher than the one it is processing, we need to intercept the current set_speed call and adjust the disk speed to the newer one. In that case, we might incur additional performance penalty while processing such interruption. In practice, this penalty rarely occurs and the effect is very minor mainly because the I/O phase in the applications we used exhibits bursty disk access patterns and each application has different length of computation (i.e., disk idle times).

5.4 Experimental Evaluation

5.4.1 Setup

We used the trace library component of Pablo [80], an I/O performance analysis toolkit, to collect the disk I/O traces for each application. We ran our applications on a Linux cluster of 16 dual processor nodes, AMD Athlon MP2000+ systems, connected through Ethernet and Myrinet. Each node of this system is configured as a PVFS2 I/O node and we installed MPICH2 in our execution environment. Each I/O node has only one disk, i.e., one disk per I/O node. All applications

\(^1\)In PVFS2, each I/O request is processed by the Trove thread spawned by the server process.
that will be explained later in this section are MPI-IO based and use PVFS2 as the file system. We compiled them using the same compilers, gcc 4.0.2 and Intel FORTRAN compiler 9.0\textsuperscript{2}, which MPICH2 was built with and ran them under the MPICH2 and PVFS2 environment.

To evaluate our approach and the prior ones to disk power management, we built a custom energy simulator. The disk energy model we used in our simulator is based on the data from the data sheets of the IBM Ultrastar 36Z15 disk [2]. Table 5.1 gives important simulation parameters. The I/O traces collected from Pablo [80] capture the duration of each I/O activity as well as the impact of slowing down or speeding up a disk. The cost of changing the speed of disk, $\Delta t$, to the different amount of RPM change, $\Delta n$, is also given in Table 5.1. We calculated the energy consumption of each application based on this timing information fed to our energy simulator. Energy consumption during each of different RPM transitions is based on the quadratic curve fitting in [22]. By default, all arrays are striped over all 8 disks.

Table 5.2 gives a brief explanation of the applications used in this study. As can be seen from this table, the applications are taken from various sources and solve different types of problems from the scientific computing domain. The last two columns give the total amount of data manipulated by each application and its base execution time, using the default values of the system parameters listed in Table 5.1 (executing each application using four processors assuming the highest speed for all disks).

To evaluate the effectiveness of our approach, we made experiments with the

\begin{table}
\centering
\caption{Default System parameters.}
\begin{tabular}{|l|c|}
\hline
Parameter & Default Value \\
\hline
Number of disks & 8 \\
Number of processors & 4 \\
Disk drive model & IBM36Z15 \\
Storage capacity (GB) & 36.7 \\
Maximum disk speed (RPM) & 15000 \\
Striping unit (Kbytes) & 64 \\
Disk layout & (0, 8, 64) \\
Available disk speeds (RPM) & (15000, 12000, 9000, 6000, 0) \\
Active power consumption (Watt) & (13.5, 10.73, 8.57, 7.03, 0) \\
Idle power consumption (Watt) & (10.2, 7.43, 5.27, 3.73, 0) \\
$\Delta t$ (sec) when $\Delta n$ (RPM) is (15K, 9K, 6K, 3K) & (10.9, 6.54, 4.36, 2.18) \\
\hline
\end{tabular}
\end{table}

\textsuperscript{2}We used the Intel FORTRAN compiler to compile the NIAL application, which is written in F90.
Table 5.2. I/O applications used in our experiment.

<table>
<thead>
<tr>
<th>Name</th>
<th>Source</th>
<th>Language/Library</th>
<th>Description</th>
<th>Data Size (MB)</th>
<th>Cycles (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MxM</td>
<td>IBM [81]</td>
<td>C/MPI-IO</td>
<td>Originally MPI-IO/GPFS test program for out-of-core block matrix multiplication. It was modified to use PVFS2 through MPI-IO, and all matrices are assumed to be square.</td>
<td>191.8</td>
<td>244.8</td>
</tr>
<tr>
<td>SSA Sim</td>
<td>Northwestern University [82]</td>
<td>C/MPI-IO</td>
<td>This is a parallel sequence search algorithm for evaluating various I/O strategies using MPI-IO. We compiled it to use collective I/O operations.</td>
<td>798.3</td>
<td>396.2</td>
</tr>
<tr>
<td>NIAL</td>
<td>MSE PSU [83]</td>
<td>F90/FFTW</td>
<td>This application quantitatively simulates phase transformation and coarsening in Ni-Al binary alloy in 3D using the KKS model using parallel programming.</td>
<td>720</td>
<td>308.9</td>
</tr>
<tr>
<td>BTIO</td>
<td>NPB2.4 [84]</td>
<td>F77/MPI-IO</td>
<td>This program implements the Block-Tridiagonal (BT) NPB problem, and employs MPI-IO. We compiled it with collective buffering.</td>
<td>419.43</td>
<td>483.9</td>
</tr>
</tbody>
</table>

following four schemes:

- **Base:** This version does not employ any disk power management strategy. All the disk energy and execution time numbers reported later in this section are given as values normalized with respect to this version.

- **COM:** This is the pure compiler-based approach proposed in [76]. In this scheme, the compiler estimates the disk idle and active periods by analyzing the application code and using the disk layouts exposed to it, and then inserts explicit disk speed-setting calls into the application. Note that the calls issued by this scheme go directly to the disk system, as opposed to our hints which go through the runtime system.

- **HW:** This is the pure hardware-based dynamic disk speed management scheme proposed in studies [10] and [22]. Depending on the observed variation on I/O response time, this scheme sets the speed of each disk to an appropriate level to save power. Note that a hardware scheme that employs spin-down disks [19, 20, 21] could also be used. However, we do not consider that option in this chapter, because the disk idle periods exhibited by these (and many other) scientific applications are very short, especially when they are executed concurrently, and consequently there are little chances of spinning the disks down and up (see Figure 5.7).
Figure 5.7. CDF (Cumulative Distribution Function) for disk idle periods in our applications. An (x, y) point on a curve indicates that y% of the idle periods have a duration of x milliseconds (ms) or lower. The minimum amount of idle time required to compensate the cost of spinning down the disk and up is called the break-even time. Based on the numbers from the IBM36Z15 [2] disk data, the break-even time is 15.19 seconds. As most of idle periods are less than 1 second, this confirms that the spin-down disk is not a viable option for these applications. Instead, multi-speed disks are more promising.

- RT: This is our runtime based approach proposed in this chapter. The compiler extracts the disk access patterns of individual application considering disk layouts and generates speed-setting hints, and then the runtime system uses these hints and current status of the shared disks to make decisions regarding appropriate disk speeds for all applications.

5.4.2 Results

Before discussing the behavior of the different schemes for reducing disk energy consumption, let us first present the I/O patterns exhibited by the applications used in this study. The graphs in Figure 5.8 present the I/O patterns (obtained using Pablo [80]) of each application in isolation. In each plot, the x-axis shows timestamps and the y-axis indicates the size of I/O requests. As we can see from these plots, each application has explicit I/O phases which can take different amount of time and use various request sizes.

We start by presenting the results for the single application execution case. Figure 5.9 gives the normalized energy consumption and execution time results
Figure 5.8. I/O request patterns of each application throughout the whole execution time. Note that, in case of BTIO, we show only the first 65 seconds of total execution time to highlight the I/O access pattern; the remaining part until the completion of BTIO shows similar I/O behavior.

(with respect to the Base scheme) when each application is executed alone. One can make several observations from the results. First, the HW scheme brings an average energy saving of 14.5% across all applications with only 4.6% performance penalty. The second observation one can make is that the COM scheme achieves a significant energy savings, 44.2% on average over the Base scheme. This significant energy saving can be attributed to the proactive (speed-setting) decisions made by the compiler. Our next observation is that the energy savings brought by the RT scheme are very similar to those obtained using the COM scheme. More specifically, the RT scheme consumes only 1–2% more energy than the COM scheme. This is mainly due to the latency overhead incurred by our runtime system in processing set_speed() calls during execution. This result indicates that our runtime system performs very well in the single application execution scenario, with only minimal
Figure 5.9. Single application execution scenario.

Figure 5.10. Multiple application execution scenario by increasing the instances of MxM up to three. The completion times for running two and three instances of MxM are 279.9 and 290.3 seconds, respectively. Due to caching effects, there is little increase in execution time when we increase the instances of MxM from two to three.

performance overhead.

In the next set of experiments we increased the number of applications running simultaneously to see how our approach behaves under multiple application scenario. Figure 5.10 shows the normalized energy consumption and execution time results when we increase the instances of MxM up to three. The results are normalized with respect to the Base case (i.e., no power management); but, in this case, the execution time represents the completion time of the last (slowest) instance (of MxM), and the energy consumption includes all the disk energy consumed by all instances (i.e., the total energy consumption in the disk system). As we can see from Figure 5.10, when we increase the instances of MxM running concurrently, the energy savings brought by the HW scheme also increases slightly whereas the
savings obtained through the COM and RT schemes decrease. The reason why
the HW scheme saves more energy with more instances is that the periodic speed
modulation by the disk drive itself comes to play with the mixed (interleaved)
patterns of I/O requests. We can also observe that the HW scheme does not incur
much performance slowdown, only 4.2% on average. However, the COM scheme
performs poorly as we increase the number of instances of MxM, in terms of both
energy and performance. Based on Figure 5.10, when three instances of MxM are
executed at the same time, the COM scheme is slightly worse than the HW scheme
in energy consumption. This is because the compiler-directed disk speed setting
calls issued by one instance of MxM can be an untimely decision (e.g., slowing
down a disk when it needs to be run at a higher speed) for the other instances of
MxM that exercise the same set of disks (recall, from Table 5.1 that, by default,
all arrays are striped over all 8 disks). The last observation we make is that the
RT scheme always achieves the highest energy savings, 30.9% on average, consid-
ering all execution scenarios. These results clearly show that our approach is very
successful with the multiple application execution scenario.

We next present results for four execution scenarios described in Table 5.3.
The second column of this table gives the applications executed simultaneously

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Running Applications</th>
<th>Completion Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>MxM</td>
<td>244.8</td>
</tr>
<tr>
<td>S2</td>
<td>MxM + S3aSim</td>
<td>464.0</td>
</tr>
<tr>
<td>S3</td>
<td>MxM + S3aSim + NIAL</td>
<td>371.9</td>
</tr>
<tr>
<td>S4</td>
<td>MxM + S3aSim + NIAL + BTIO</td>
<td>651.4</td>
</tr>
</tbody>
</table>

Figure 5.11. Multiple application execution scenario with different set of applications.
in the corresponding scenario and the third column gives the completion time, the time it takes for all the applications in the scenario finish their execution (under the Base scheme). As we can see from this table, the completion times for these different scenarios increase significantly when we execute more applications in parallel. Figure 5.11 shows the effect of three schemes we experimented with the scenario in Table 5.3 on the energy and execution times. In this figure, the bars for a given scenario is normalized with respect to the energy and execution times of the Base scheme. We can observe similar trends from these results as in the cases where we execute the multiple instances of the same application (see Figure 5.10). Specifically, while the HW scheme achieves increased energy savings when the number of applications is increased, the COM scheme does not perform well in saving disk energy. These results are expected because, from the compiler perspective, increasing the number of applications running in parallel has an impact on the performance and, ultimately, on the disk energy consumption, regardless of whether the instances running in parallel belong to the same application or not. In this execution scenario, the RT scheme achieves 19.4% and 39.9% energy savings over the HW and the COM schemes, respectively. As far as the performance is concerned, the RT scheme degrades the base execution time by less than 7.6%. The HW and COM schemes, on the other hand, incur 16.7% and 23.1% performance slowdowns, respectively. Note that the performance of the HW scheme under this execution scenario is much worse than the execution under the multiple instances of MxM (see Figure 5.11). This is because the heuristic employed in the HW scheme modulates the disk speed based on the number of request (rather than time), and this modulation heuristic incurs more performance penalty when the number of I/O requests are increased.

In the last set of experiments, we varied the default striping of each file to assess the impact of different disk layouts on performance and energy consumption. In these experiments, we used the S4 scenario (see Table 5.3) only. Recall that, all the results presented so far are obtained using 8 disks and all files are striped across all available disks. In this new set of experiments, we increased the total number of disks that can be used for file striping to 15, and varied the Base disk for each array to effect its disk layout (see the disk layout triplet discussed in Section 5.2). The other two parameters, Striping_Factor and Striping_Unit,
We used all four applications, each of which is configured to get striped across eight disks among fifteen available disks. The file sizes are fixed at 8 and 64Kbytes, respectively, as in the initial setup. We generated a random number between 1 and 15 to select the Base disk from which the file striping starts. Figure 5.12 gives the normalized energy consumption and execution time results for the S4 scenario with three different disk layouts. We see that the overall energy savings brought by our runtime system based approach are similar across the different layouts. This indicates that our approach can adapt very well to the different disk layouts. This is mainly because the compiler can extract the disk access patterns accurately with different disk layouts and provide the most preferable decision considering the underlying disk layouts. Our runtime system takes these preferable speeds and makes a globally acceptable decision for each and every disk. To summarize, these results show that our approach works very well when disks are shared and when they are not shared.

5.5 Effect of Caching

While our experiments, so far, focused on systems that do not employ any explicit file caching mechanisms, in this section, we also performed experiments with a system with a caching configuration. Note that, our I/O traces collected so far already reflect caching effect within a processor node and do not include any unnecessary I/O requests to the disk subsystem, as they are gathered on a real machine as a result of real execution. In this section, we want to quantify the effect...
of file caching to the behavior of I/O access pattern that might affect the energy consumption.

Traditionally, the I/O caching has a significant influence on I/O performance. While the LRU (Least Recently Used) replacement policy, which dates back at least to 1965 [85], has been widely used to manage buffer caches, there are various approximations and enhancements to this, for example, the classical CLOCK algorithm [86]. While variants of the CLOCK algorithm are still dominant in many implementations, they also have some limitations in adaptability as far as changing access patterns is concerned. To mitigate this problem, several researchers studied enhancements to the classical CLOCK algorithm, such as 2Q [87] and LRFU [88]. More recent studies that try to handle accesses with weak temporal or spatial locality include CAR (Clock with Adaptive Replacement) [89], LIRS (Low Inter-reference Recency Set) [90], ARC (Adaptive Replacement Cache) [91], CLOCK-Pro [92], Second-Tier Cache Management [93], MultiQueue [94], and DULO (DUal LOcality) [95]. Patterson et al [68] used a hint mechanism, which is designed to expose access patterns to the runtime system, in managing prefetching and caching file cache blocks. Targeting multi-level caches which are quite common in modern systems, several multi-level buffer cache management policies have been proposed [96, 97, 98, 99]. The main idea behind these approaches is to manage cache blocks in an exclusive manner, that is, a file cache block in one level of cache hierarchy does not present in another level of caches. While it is possible to employ several buffer cache management policies, in this section we use the LRU policy which replaces the least recently used file stripe when a new block is to be brought in.
In this experiment, we implemented a buffer cache simulator and the traces collected earlier are fed to our cache simulator. We present only the energy results for a single application execution with the RT scheme; we expect similar trends with other schemes. The normalized energy consumption results given in Figure 5.13 indicate that the results obtained using a system with a cache are comparable to those obtained in a cacheless system. Note that a buffer cache can influence energy savings due to disk power management as it modifies the number of accesses to the disk subsystem. Typically, one would expect a smaller number of disk accesses with larger buffer caches. The results with “No cache” are regenerated from the ones in Figure 5.9(a). One can make several observations from these results. First, NIAL and S3aSim do not generate any additional energy savings regardless of the cache size we experimented. This is because these two applications are write-oriented. Therefore, all the I/O requests need to be translated to the disk accesses. Second, MxM generates a small improvement in energy savings as we increase the size of buffer cache because one of input matrices exhibits certain amount of data reuse. However, the energy savings due to caching are limited because not all I/O requests in each I/O phase cannot be eliminated. Lastly, we can save 18% more energy consumption for BTIO when the cache size is 512MB. Note that the purpose of I/O by BTIO is also for checkpointing, i.e., all I/O is write operation. Unlike the other two write-oriented applications, NIAL and S3aSim, BTIO has a read phase toward the end of execution for verification of the data written to the disk during checkpointing. We can eliminate all disk accesses during this read phase provided that the amount of buffer cache size is larger than the amount of data written during the write phase. Since the data size of BTIO we experimented is about 420MB, the effect of caching realized when the buffer cache size employed is more than that.

5.6 Summary

This chapter proposes a runtime system support for software-based disk power management in the context of MPI-IO based scientific applications. In the proposed approach, the compiler provides important information on future disk access patterns and preferable disk speeds for each MPI-IO call. The proposed runtime
system uses this information to make decisions (regarding disk speeds) that would be agreeable to all applications running concurrently. Our goal is to save as much energy as possible without slowing down any application significantly. We also quantify the effect of file caching to overall energy consumption.
Conclusions

6.1 Summary of Major Contributions

Power consumption by large servers and clusters has recently been a popular research topic, since this issue is important from both technical and environmental viewpoints. 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 dissertation proposes and experimentally evaluates a software-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.

More specifically, this dissertation presents our contributions to conserve disk power consumption through the support of compiler and runtime system, which are listed as follows:
• It presents a compiler-based approach to disk power management for data-intensive scientific applications. In this approach, the compiler derives data access pattern and, by combining this information with disk layout of array data, it obtains the disk access pattern. This extracted information then can be used to make proactive disk power management decisions.

• It presents three code transformation and restructuring schemes to improve the effectiveness of underlying disk power management schemes.

• It presents a compiler-directed energy-aware prefetching scheme for multi-speed disks. The proposed approach determines the prefetch distances for disk access instructions, the disk speeds for all disks in the disk subsystem, and the data layouts on the disk in a unified manner.

• It proposes a runtime system support within PVFS2 for software-based disk power management in the context of MPI-IO based scientific applications.

Our experimental results indicates that we achieve 39.9% and 19.4% savings in disk power consumption, over the pure hardware based and the pure compiler based schemes, respectively.

6.2 Ongoing Work

I/O prefetching has been employed in the past as one of the mechanisms to hide large disk latencies [64, 18, 100, 101, 102, 68, 69, 103, 104]. However, I/O prefetching in parallel applications is problematic when multiple CPUs share the same set of disks due to the possibility that prefetches from different CPUs can interact on shared memory caches in the I/O nodes in complex and unpredictable ways. To address this problem, we (i) quantify the impact of compiler-directed I/O prefetching – developed originally in the context of sequential execution – on shared caches at I/O nodes. The experimental data collected shows that while I/O prefencing brings benefits, its effectiveness reduces significantly as the number of CPUs is increased; (ii) identify inter-CPU misses due to harmful prefetches as one of the main sources for this reduction in performance with the increased number of CPUs; and (iii) propose and experimentally evaluate a profiler and compiler assisted adaptive
I/O prefetching scheme targeting shared storage caches. The proposed scheme obtains inter-thread data sharing information using profiling and, based on the captured data sharing patterns, divides the threads into clusters and assigns a separate (customized) I/O prefetcher thread for each cluster. In our approach, the compiler generates the I/O prefetching threads automatically.

I/O prefetching also requires an accurate timing to be effective in practice since early and late prefetches do not bring much benefit and can in some cases be even harmful. This timing problem is particularly problematic when multiple CPUs share the same set of disks due to the possibility that prefetches from different CPUs can interact on shared memory caches in the I/O nodes in complex and unpredictable ways. To solve this problem, we (i) quantify the impact of compiler-directed I/O prefetching on shared caches at I/O nodes. The experimental data collected shows that while I/O prefetching brings some benefits, its effectiveness reduces significantly as the number of CPUs is increased; (ii) identify inter-CPU misses due to harmful I/O prefetches as one of the main sources for this reduction in performance with increased number of CPUs; and (iii) propose and experimentally evaluate prefetch throttling and data pinning schemes to improve performance of I/O prefetching. Prefetch throttling prevents one or more CPUs from issuing further prefetches if such prefetches are predicted to be harmful, i.e., replace from the memory cache the useful data accessed by other CPUs. Data pinning on the other hand makes select data blocks immune to harmful prefetches by pinning them in the memory cache. We show that these two schemes can be applied in isolation or combined together, and they can be applied at a coarse or fine granularity.

6.3 Future Research Directions

New demands have arisen for networked storage system and storage processors. Typical services a storage processor performs include compressing data, encrypting data and scanning data to detect malware. Another crucial service for modern storage systems is storage virtualization. Parallel to this, multi-core processors are ubiquitous these days, even in storage processors. The multi-cored storage processors are relative new, but the real challenge is how to fully utilize the computing powers present in those processors while meeting new demands for networked stor-
Our long-term goal is to investigate the following critical questions when designing a high-performance, energy-efficient storage and file systems. (1) How can we exploit multi-core capabilities in both storage processors and computing processors?; (2) How can we make systems secure by providing storage system level encryption?; (3) How can we enhance I/O performance in large-scale storage systems?; and (4) How can we virtualize storage while balancing performance requirements and energy efficiency in networked storage systems?

As systems’ sizes and capabilities approach the petascale range, the I/O and storage layers are becoming extremely critical; but they also have become a bottleneck to sustaining scalable performance. I/O caching has been popularly employed in the past to improve I/O latency for many data-intensive applications [85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 105]. Almost all high-end clusters and servers nowadays are equipped with hundreds of thousands of processors, which are interconnected through a particular network topology. Most of the prior I/O caching schemes, however, are oblivious to this network architecture. In the near future I plan to investigate an I/O caching scheme that takes the network topology into account when designing caching policies. The idea is to determine, for each data block, the most appropriate cache in the architecture in which to store the copy at any given point during the course of execution. This can be accomplished, for example, by formulating the problem as a post-office placement problem. Our long-term goals are to research efficient caching and prefetching schemes, and to design topology-aware collective operations in both communication and I/O schemes that take advantage of state-of-the-art network architecture and multi-core architectures on computing nodes.

Lastly, in our future work, we would like to investigate energy-efficient collective I/O operations [106, 107, 108]. Most large-scale parallel systems provide some sort of collective I/O operations to meet the I/O requirements of I/O-intensive applications. The implementation of such collective I/O operations typically involves I/O operations followed by communication or vice versa. One important characteristic of these collective I/O operations is that, in order to make sure correct data communication before and after I/O operations, there must be a synchronization point between I/O and communication. Due to the collaborative processing of I/O
and communication in collective I/O and the load imbalance in computing node, not all I/O operations terminate at the same time before/after communication. We would like to exploit these properties in collective I/O operations to minimize disk energy consumption.


Vita
Seung Woo Son

Seung Woo Son is a Ph.D candidate in the department of computer science and engineering at The Pennsylvania State University. He received his BS and MS, both in computer engineering, from Yeungnam University, Korea, in 1995 and 1997, respectively. From 1997 to 2003, he was a research engineer in the embedded software center at ETRI (Electronics and Telecommunications Research Institute), one of national research laboratories in Korea. His research interests include compiler optimizations for storage systems, power-aware computing, and embedded software. He is IEEE and ACM student member.