Energy-Aware Data Prefetching for Multi-Speed Disks*

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

ABSTRACT

Power consumption of disk based storage systems is becoming an increasingly pressing issue for both commercial and scientific application domains. Prior work proposed several hardware based approaches to reducing disk power consumption by making use of techniques such as spinning down idle disks and rotating them at lower speeds than the maximum speed possible. While such techniques are certainly very important, it is also critical to consider the influence the software can exercise in shaping the power consumption behavior of disk-intensive application programs. Motivated by this, the main goal of this work is to study whether an optimizing compiler can be used for increasing the power benefits that could be obtained from multi-speed disks. Specifically, we propose and experimentally evaluate a compiler-directed energy-aware data prefetching scheme for scientific applications that process disk-resident data sets. This scheme automatically determines the prefetch distance for all disk access instructions, the disk speeds to be employed, and the associated disk layouts (striping parameters) in a unified setting. We implemented the proposed approach within an optimizing compiler framework and conducted experiments with several disk-intensive applications. Our experimental evaluation shows that the proposed approach brings significant reductions in disk energy consumption over a state-of-the-art software-based I/O prefetching mechanism that does not take into account energy consumption explicitly. Our results also show that the energy-aware prefetching scheme does not bring any extra performance penalties and the energy reductions achieved are consistent across a wide spectrum of values of the simulation parameters.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

General Terms

Algorithms, Design, Performance, Experimentation

*This work is supported in part by NSF grants #0444158 and #0406340, and a grant from GSRC.

CF'06, May 3–5, 2006, Ischia, Italy.

Keywords

Optimizing Compiler, Prefetching, Multi-Speed Disks, Low Power

1. INTRODUCTION

High power consumption is one of the most pressing issues for computing platforms that target large-scale data-intensive applications [7, 6, 12, 13]. While most of the recent research efforts on minimizing power consumption have been performed in the CPU, network, and memory domains, the research on disk power optimization is still in its infancy. A couple of recent papers (e.g., providing multi-speed setting for server disks [5, 14], power-aware storage cache management schemes [32, 33], and compiler-guided disk power management schemes [25]) have focused exclusively on disk power consumption and proposed hardware and software based solutions to the problem. Most of these papers estimate and/or control disk power consumption or present static/dynamic code/data reorganizations for maximizing power savings that could be obtained from the low-power operating modes supported by the disk system.

While conventional disk power optimization approach [11, 10, 18] 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 [5, 14] 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 [19] and [30]) 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 main goal of this paper 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 paper 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.

We implemented the proposed approach within a research compiler [15] and made experiments with four different data-intensive applications that process disk-resident datasets. The results from our experiments indicate that the proposed energy-aware I/O prefetching approach reduces disk energy consumption over a state-of-the-art, energy-agnostic I/O prefetching scheme by 19.6% on average, without hurting the performance of the latter. Our experimental results also show that the achieved disk energy savings are consistent across a wide range of values of the major simulation parameters, and that our approach introduces very little (less than 1%) performance overhead, as compared to the conventional I/O prefetching.

The remainder of this paper is structured as follows. The next section discusses the related work on disk power optimization. Section 3 gives a high level view of the storage system under consideration and defines the technical concepts frequently used in this paper. Section 4 gives an example to demonstrate the benefits of the proposed approach. Section 5 explains the technical details of our approach. An experimental evaluation of our approach and its quantitative comparison against the prior work are presented in Section 6. Finally, the paper is concluded by a summary and a brief discussion of the planned future work in Section 7.

2. RELATED WORK

Most of the prior studies on reducing disk power/energy consumption make use of observed idle times during program execution. To exploit disk idle periods, the disk drive itself needs to provide a low-power operating mode, either in the form of completely stopping disk rotation (spinning down) or in the form of dynamically adjusting the rotational speed. Providing low-power modes is important, because even if a disk is idle it consumes almost as much energy as it would consume in the active (fully operational) mode [14, 17]. For the laptop/desktop domain where the applications typically exhibit long idle periods, several studies have already considered techniques such as spinning down idle disks by using a fixed threshold period (i.e., the time to wait before spinning down a disk) or by estimating the threshold period adaptively [10, 11, 18].

Once the disk is equipped with some sort of low-power operating mode, we can make use of these modes within an operating system (OS) or at an application level by increasing the duration of idle periods so that a given disk can be placed into low-power modes for longer durations of time. Among the efforts focusing on the OS layer, Zhu et al [32] and Papathanasiou et al [20] consider power-aware caching and prefetching strategies. The rationale behind both these studies is that conventional I/O caching and I/O prefetching techniques, which mainly focus on the performance angle, can hardly produce any long idle periods. Rather than spreading disk accesses across the entire execution period, energy-efficient prefetching generates burst disk access patterns, which is preferable from the energy perspective. The enlarged idle periods in turn allow a disk to be placed into one of the supported low-power operating modes.

Zhu et al [32] also study a power-aware cache replacement algorithm, called PA-LRU, in the context of large storage systems, which are typically equipped with several GBs of aggregated cache memory. The main idea behind their approach is to selectively maintain cache blocks from certain disks, so that the remaining disks can stay in low-power modes for a longer period of time. In another paper, Zhu et al [33] propose a different approach, called PB-LRU (Partition-Based LRU), to the same problem. PB-LRU explores various cache replacement techniques in the context of disk arrays equipped with multi-speed disks. Lastly, Zhu et al [31] recently proposed a holistic disk power management technique, called Hibernator, that combines three major techniques: dynamic disk speed setting, multi-tier data layout, and data reorganization. Since frequent modulations of disk speeds might decrease disk reliability, their idea is to adjust disk speed at a coarse granularity. To guarantee the specified response time limit, Hibernator keeps track of average response time dynamically. If the specified response time guarantee is at risk, Hibernator restores the speeds of all disks to full speed.

Several studies investigated the problem of disk power management at the application/compiler level. For example, Heath et al [16] studied an application code transformation technique for energy-aware device management by generating I/O burstiness in laptop disks. More recently, Son et al proposed several compiler-based code transformation techniques to conserve disk energy consumption. First, they studied a compiler technique that inserts explicit disk power management calls in sources codes of scientific applications [25]. The idea is that a compiler can extract how disks are traversed during execution time using the application source code along with the file level striping information. By inserting explicit power management calls, e.g., spin_up and spin_down, in the application code, one can eliminate (to a large extent) the performance penalty that would normally be incurred by reactive disk power management schemes. Second, they revisited conventional loop distribution and iteration space tiling techniques from an energy perspective. To achieve the best energy savings without slowing down performance much, they showed that both code and underlying disk layout must be considered at the same time. In another paper [24], the same authors described a compiler approach to reduce disk power consumption in the presence of parallel disk systems. To increase disk idleness, the proposed technique schedules the code fragments assigned to a number of processors according to the disk access patterns extracted by an optimizing compiler, which captures both intra- and inter-processor disk reuses.

Since large data centers host huge amounts of data for several application domains, they typically exhibit locality at a disk partition level or a file level. This means that, in a given time period, not all the disks participate in servicing I/O requests. Observing this, MAID (Massive Arrays of Idle Disks) [9] was proposed to reduce disk energy consumption using a small number of disks as cache drives, thereby potentially reducing the number of spin-ups for disks. While cache drives service the requests to the disk array, other unused disk drives can be placed into the low-power modes. Pinheiro et al [21], on the other hand, proposed a data migration technique called PDC (Popular Data Concentration). The main idea behind this scheme is to dynamically move the most frequently-accessed disk data to a subset of the disks in the array, thereby increasing the idle periods for the remaining disks in the system. PDC is a feasible solution for network servers because workloads processed by such systems are heavily skewed towards a small set of files. Techniques such as MAID and PDC manipulate data at a file system granularity, therefore, at least one day is required to collect the file access patterns and adjust the file layouts according to the gathered information.

The approach proposed in this paper is different from all pure hardware based disk power management schemes since it is compiler based. It is also different from the prior compiler based studies in that, it minimizes disk energy consumption through code hoisting (energy-aware prefetching), instead of linear code transformations. In addition, as against studies such as [32], our approach determines the prefetching distance, disk speeds, and data layouts

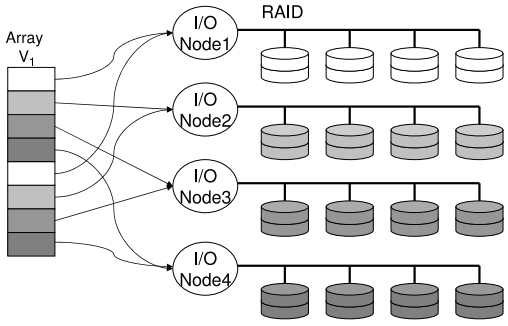


Figure 1: Two-level striping of array data across disks.

in a unified setting. However, we also want to mention that the approach proposed in this paper can also be used in conjunction with prior compiler-directed code modification schemes such as [24] for reducing disk power consumption even further. Finally, in contrast to the previous studies that target multi-speed disks, our approach determines disk speeds statically for each application at compile-time. Therefore, it has practically no impact on reliability due to the frequent modulation in disk speed at run-time.

3. HIGH LEVEL VIEW OF STORAGE SYSTEM

The storage system considered in this work is shown in Figure 1 at a high level. Our focus is on large, data-intensive scientific applications that manipulate disk-resident, multi-dimensional arrays. The disk requests in this architecture are directed to I/O nodes over which the array files are striped. Within each I/O node, a stripe assigned to that I/O node is further striped at the RAID level (depending on the specific RAID implementation [8] adopted). Therefore, as depicted in Figure 1, each data array in our storage architecture is striped at *two different levels* (I/O node level and RAID level). While the RAID level striping is hidden from the software, the I/O node level striping is visible to the software (to the compiler in our case) and can be controlled through calls from the underlying I/O library and/or the parallel file system used. For example, in PVFS [4, 23], one can manipulate the I/O node level striping information of files by changing the `pvfs_filestat` structure, which includes the stripe unit and the number of disks used for striping.

In this paper, we determine rotational speeds of disks and data layouts of arrays at an I/O node granularity. That is, when we set the speed of a particular I/O node, it means setting the speed of all the disks controlled by that I/O node. However, for the ease of discussion, we use the term "disk" instead of "I/O node" when we explain our approach below. The "disk layout" concept used in the rest of this paper refers to the I/O node level striping; i.e., when we mention "striping", we mean the striping at the I/O node level. In our experimental evaluation, we assume a one-to-one mapping between data arrays and files. In other words, we assume that each data array is stored in a single file and a file contains only a single array. Under this assumption, one can talk about "striping an array over the I/O nodes." While we can easily relax this assumption by allowing one-to-many and many-to-one mappings between the disk files and the data arrays, we do not evaluate these options in this paper.

The proposed compiler-directed approach operates under two assumptions. The first assumption is that the I/O node level striping can be accessed and controlled by the compiler. This is possible because current parallel file systems and run-time libraries (e.g., PVFS [4, 23]) provide API calls that enable this. Our second assumption is that the disk system is exercised by a single application at a time (of course, the different applications can use the same system at different times). In our approach, the compiler can manage/control the disk power consumption by inserting prefetching instructions to array data, which are stored in multi-speed disks. Since storing array data in a low-speed disk does *not* destroy the data itself, our approach will *not* create a correctness issue if the second assumption fails. However, if the disk speeds determined when considering one application are not appropriate for the other concurrently-executing applications, our energy savings might be reduced and we can incur I/O performance degradations unless we tune the disk speed for the other applications accordingly. We believe that the disk usage information extracted by our compiler can be passed to the OS at specific program points, and the OS in turn can use this information to implement a global disk power management algorithm. However, such extensions are not the focus of this paper. Our goal instead is to evaluate the potential power savings from a single application's viewpoint when energy-aware prefetching is employed.

4. MOTIVATIONAL EXAMPLE

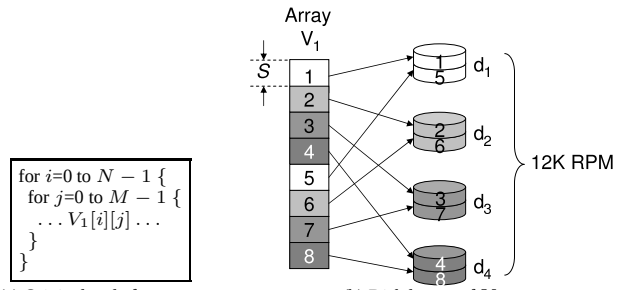
In this section, we demonstrate how our approach can reduce disk energy consumption by hiding latencies of low-speed disks using the example code fragment shown in Figure 2(a). The code fragment given in this figure accesses a two-dimensional disk-resident array, named V_1, using a loop nest constructed from two loops. For illustrative purposes, V_1 is assumed to be striped over 4 disks with a stripe size of S (see Figure 2(b)) and all four disks in question are assumed to be running at 12,000 RPMs. As depicted in Figure 3(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, we 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.

Since our approach 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 [2] to hide disk I/O stall time and reduce overall execution latency. The code fragment after applying I/O prefetching is given in Figure 2(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:

$$d = \lceil \frac{T_d}{s + T_{pf}} \rceil, \quad (1)$$

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 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 2(c). In this example, d iterations of j loop are assumed to be required to hide I/O latency and $b1$ 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 over-



(a) Original code fragment. (b) Disk layout of V_1

```
for i=0 to N − 1 {
 PF (&V1[i][0]); /* prolog */
 for jj=0 to M − 1 − d, step b1 { /* steady-state */
  PF (&V1[i][j + d]);
  for j=jj to jj + b1 {
   . . . V1[i][j] . . .
  }
 }
 for j=M-d to M − 1 { /* epilog */
  . . . V1[i][j] . . .
 }
}
```

(c) Code with prefetching. d is the prefetch distance.

Figure 2: An example application of prefetching.

head. 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 [14]. 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 3 and Figure 4 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 3(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 (1) given above. On the other hand, the energy consumption profiles after applying prefetching with different prefetch distances are depicted in Figure 4. Figure 4(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(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(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.

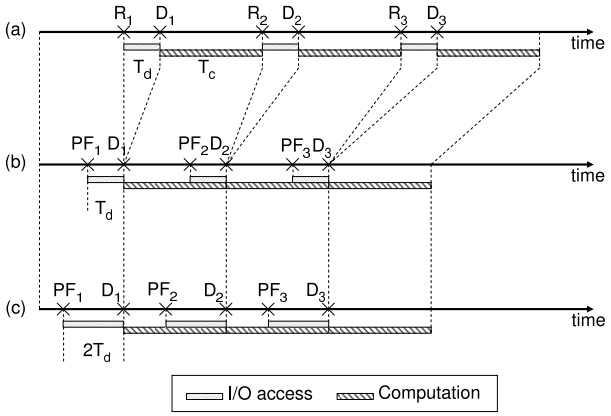


Figure 3: Comparison of I/O timings. (a) Original code without prefetching. (b) Prefetching to high-speed disks. (c) Prefetching to low-speed disks. T_d is the disk I/O time for a single block data.

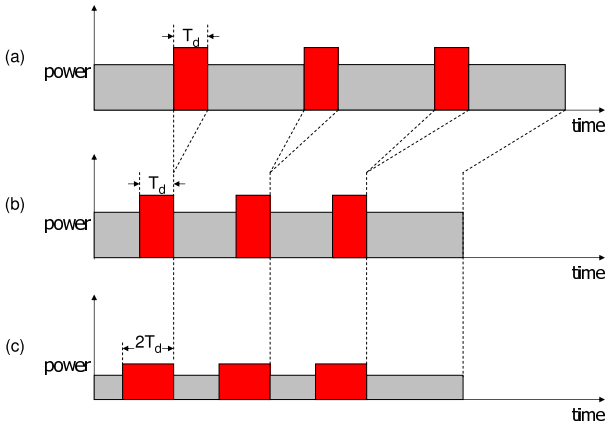


Figure 4: Comparison of disk power states. (a) Original code without prefetching. (b) Prefetching to high-speed disks. (c) Prefetching to low-speed disks. T_d is the disk I/O time for a single block data.

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.

5. COMPILER ALGORITHM

In this section, we discuss the details of our compiler algorithm for energy-aware prefetching that determines prefetch distance, disk speeds, and data layouts on disks (I/O nodes).

5.1 Basics

Before describing the algorithm, let us first define a few important mathematical concepts. In our framework, an array based, loop-intensive program $\mathcal{P}$ that consists of s loop nests is represented as:

$$\mathcal{P} = (\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_s),$$

where $\mathcal{L}_i (i = 1, 2, \ldots, s)$ is the ith loop nest in program $\mathcal{P}$. We further assume that a loop nest $\mathcal{L}_i$ of the following form:[1]

$\mathcal{L}_i$: for i_1 = l_1 to u_1, step b_1
  for i_2 = l_2 to u_2, step b_2
    ...
      for i_k = l_k to u_k, step b_k
        {loop body}

can be represented as:

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

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 jth array reference in the body of loop nest $\mathcal{L}_i$. While executing, loop nest $\mathcal{L}_i$ is assumed to access n arrays, V_1, V_2, ..., V_n. We use $\mathcal{V}$ 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 $V_i[\vec{F}(\vec{I})]$ $(i = 1, 2, \ldots, n,\ j = 1, 2, \ldots, m)$, where V_i is the name of the array and function $\vec{F}$ maps iteration vector $\vec{I}$ to a vector of subscripts for array V_i. Specifically, $\vec{F}(\vec{I})$, 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:

$$\vec{F}(\vec{I}) = M\vec{I} + \vec{o},$$

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 [28].

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, we define the disk-layout for each array (V_i) using a triplet of the following form:

$$(start\_disk, stripe\_factor, stripe\_size),$$

where start_disk is the first disk where the file striping starts from, stripe_factor is the number of disks being used for striping, and stripe_size is the unit size of each file stripe residing on each disk. For example, the layout of array V_1 in Figure 2(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.

5.2 Energy-Aware Prefetching

To exploit low-speed disks using prefetching in order to save energy, our prefetching algorithm needs to analyze the data locality exhibited by each loop nest $\mathcal{L}_i$ in program $\mathcal{P}$. Given the mathematical representation discussed in Section 5.1, temporal reuse is said to occur between two loop iterations $\vec{I}_1$ and $\vec{I}_2$ whenever $\vec{F}(\vec{I}_1) - \vec{F}(\vec{I}_2) = \vec{0}$. That is, temporal reuse occurs whenever the difference between the two loop iterations lies in the nullspace of $M(\vec{r}) = \vec{0}$, i.e., $span(M)$. Spatial reuse, on the other hand, is said to occur when two different loop iterations access the same row (in a given array) [28]. 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 $span(M_S)$. 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* [28]. This space captures only those loops for which data reuse can result in data locality. In our 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 [2]. 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.

Using the obtained the vector space representation of data locality exhibited by each loop $\mathcal{L}_i$, our approach next determines prefetch distance (d value in Equation 1) for each array reference ($V_i[\vec{F}(\vec{I})]$ made by the loop body of nest $\mathcal{L}_i$. Note that, once the d value is calculated and reference $V_i[\vec{F}(\vec{I})]$ is found to have spatial locality on ith loop, the ith 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 ith loop that changes the value of the array-indexing function ($V_i[\vec{F}(\vec{I})]$). 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 (1)), while power consumption is quadratically reduced by the amount of disk speed scaling [14]. Therefore, we need to tune the prefetch distance based on the disk speed, and in fact, our approach determines them together, as explained below.

In the first step of our 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 our approach, we determine the preferable disk 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 5. The for-each loop in this algorithm goes over the loop nests in the

[1] If $\mathcal{L}_i$ is not perfectly nested, one can use techniques such as code sinking [29] to make it perfectly nested.

[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, ..., L_s);
   Available disk speeds, RPM = (1, 2, ..., l);
OUTPUT:
   Determined RPM-group(i), where 1 ≤ i ≤ l;

T_pf = the number of cycles for PF instruction;
for each V_k ∈ V // for each array;
    G[V_k] = ∅; // possible disk speeds for each array;

// repeat for each loop nest L_i.
for each L_i ∈ 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 L_i
        // assume that a_i(Ī) accesses array element V_k[F̄(Ī)].
        for each array reference a_i(Ī) {
            calculate I/O latency, T_d(j), when RPM is j;
            // determine prefetch distance, d_j, at jth RPM.
            d_j = ⌈T_d(j) / (s_i + T_pf)⌉;
            if (d_j > total number of iterations for the pipeline loop)
                G[V_k] = G[V_k] ∪ {j};
        }
    }
}

// RPM-group(l) generated by adding maximum value from set G[V_i].
for each array V_i {
    l = {x|x ∈ G[V_i] and MAX(G[V_i])};
    RPM-group(l) = RPM-group(l) ∪ {V_i};
}
```

Figure 5: Disk speed detection algorithm.

```
INPUT:
   Input program, P = (L_1, L_2, ..., L_s);
   Determined RPM-group(i), where 1 ≤ i ≤ l;
OUTPUT:
   Determined data layout for each array;

tot_disks = total number of disks available;
init_disk = 0;
weight[V_i]: the number of accesses made to V_i within P;
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 ∈ RPM-group(i)
        sum += weight[V_i];
    stripe_factor(V_i) = tot_disks × ⌈sum / weight[V]⌉;
    tot_disks –= 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 {
    start_disk (V_i) = init_disk;
    init_disk += stripe_factor (V_i);
}
```

Figure 6: Data layout detection algorithm.

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, our approach 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 our approach, we determine the disk layouts of the arrays in the application. In order to do this, we first form what we call the *RPM-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. Our 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 3) 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 our 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 6.

The last step of our approach is to restructure the application code 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 [2]. Figure 7 shows the pseudo-code for the algorithm that modifies the application code. The overall view of our approach to energy-aware prefetching is depicted in Figure 8.

As explained above, our 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, our 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 9(a). As shown in Figure 9(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 [1, 22], 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 9(c). This small example shows that our approach can be applicable even if the data layouts and the disk speeds are determined a priori.

5.3 Example

We now give a more detailed example to show how our algorithm described in Section 5.2 works in practice. The original code fragment shown in Figure 10(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 $\mathcal{P}$, as shown in Figure 10(b). This locality information indicates that, in the first loop nest ($\mathcal{L}_1$), all three array

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:
Transformed 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 II_p (=$[l'_p, u'_p]$) to the loop nest $\vec{I}$
such that $\vec{I'} = (I_1, \ldots, II_p, I_p, \ldots, I_k)^T$;
// calculate new loop bounds for II_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 "$a_1(\vec{I'}), \ldots, a_m(\vec{I'})\rangle$";
emit "$\rangle$";

Figure 7: Code restructuring algorithm.

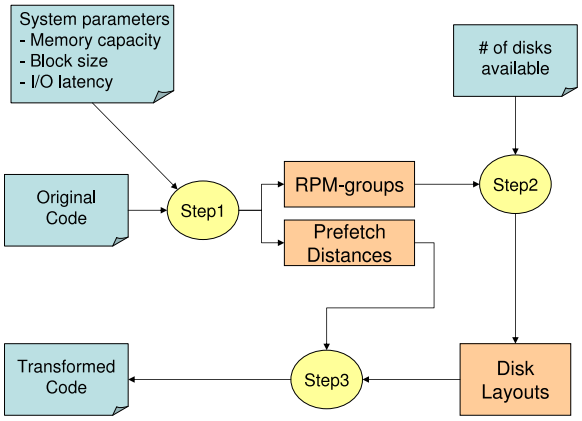
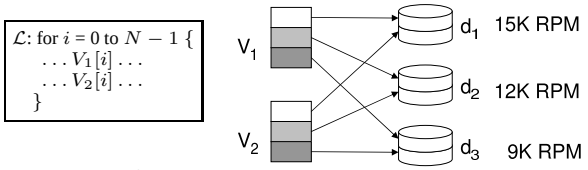


Figure 8: The three steps of our approach to energy-aware data prefetching.

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 5). 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 $\mathcal{L}_1$ 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 $\mathcal{L}_2$ and $\mathcal{L}_3$ respectively, possible disk speeds for V_1 and V_2 arrays are also dependent on the $\mathcal{L}_2$ and $\mathcal{L}_3$ 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 10(c). This indicates that, for the array V_1 accessed by both $\mathcal{L}_1$ and $\mathcal{L}_2$,



(a) Original code fragment. (b) Data layouts and disk speeds.

$\mathcal{L}_1$: for ii = 0 to $\frac{(N-1)}{3} - d_1$, step b_1 {
PF (&$V_1[i + d_1]$); PF (&$V_2[i + d_1]$);
for $i = ii$ to $ii+b_1$ {
. . . $V_1[i], V_2[i], \ldots$
}
}

$\mathcal{L}_2$: for $ii = \frac{(N-1)}{3}$ to $\frac{2(N-1)}{3} - d_2$, step b_2 {
PF (&$V_1[i + d_2]$); PF (&$V_2[i + d_2]$);
for $i = ii$ to $ii+b_2$ {
. . . $V_1[i], V_2[i], \ldots$
}
}

$\mathcal{L}_3$: for $ii = \frac{2(N-1)}{3}$ to $N - 1 - d_3$, step b_3 {
PF (&$V_1[i + d_3]$); PF (&$V_2[i + d_3]$);
for $i = ii$ to $ii+b_3$ {
. . . $V_1[i], V_2[i], \ldots$
}
}

(c) Transformed code fragment. The three loops show only the steady-state of the pipelined loops.

Figure 9: Application of our approach when data layouts and disk speeds are not adjustable.

we can assign 12K RPM because the j loop in $\mathcal{L}_2$ 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 $\mathcal{L}_3$ indicates that there is no inherent spatial locality. Based on these disk speeds determined, the resulting data layouts (determined using the algorithm in Figure 6) and the transformed code fragment (obtained using the algorithm in Figure 7) are given in Figures 10(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 10(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 [2], that does not care about energy consumption.

6. EXPERIMENTAL EVALUATION

6.1 Simulation Platform

To evaluate the effectiveness of our approach in reducing disk energy consumption, we implemented a simulation platform using DiskSim [3]. 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 [17] and its relevant power and performance characteristics are shown in Table 1. Since we use multi-speed disks running at dif-

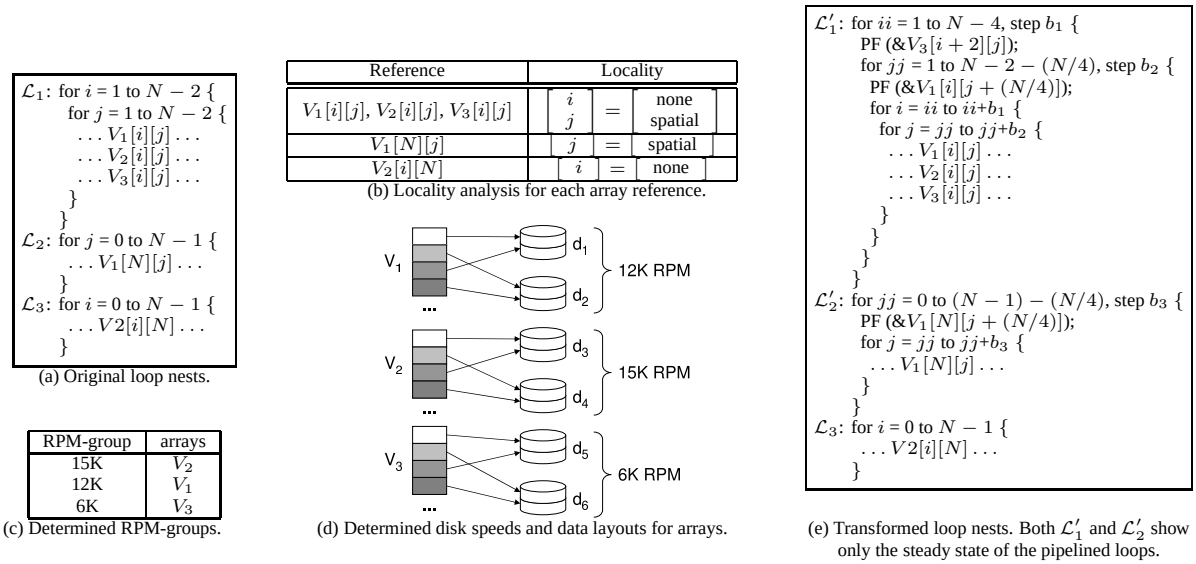


Figure 10: An example application of our hoisting algorithm.

ferent 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 [14]. The energy and performance values for these multi-speed disks are also given in Table 1.

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

- 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 2).

- PF: This scheme corresponds to the conventional I/O prefetching approach, as explained in [2]. 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 5. 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 [26]. The important characteristics of these benchmark programs are given in Table 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 1, the baseline energy and performance results are given in the last two columns of Table 2. These baseline results are obtained by executing our benchmark programs on a disk subsystem 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.

6.2 Results

The bar-chart shown in Figure 11 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 12. 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

Table 1: Major simulation parameters and their default values.

Disk Parameters		Disk Performance and Energy Model		Striping Information	
Parameter	**Value**	**Parameter**	**Value**	**Parameter**	**Value**
Disk Model	IBM Ultrastar 36Z15	Rotation speed	15K/12K/9K/6K RPM	Stripe size	64 KB
Interface	SCSI	Average seek time	3.4/3.76/7.0/10.83 ms	Stripe factor (# of disks)	16
Storage Capacity	18.4 GB	Average rotational latency	2.0/2.5/3.33/5.0 ms	Starting iodevice (starting disk)	1 (the first disk)
Disk Cache Size	4 MB	Power (active)	13.5/11.3/9.1/6.9 W		
Internal Transfer Rate	55 MB/sec	Power (idle)	10.2/8.66/7.12/5.58 W		

Table 2: Benchmarks and their characteristics.

Name	Description	Data Size (GB)	Base Energy (J)	Exec Time (sec)
171.swim	Shallow Water Modeling	115.2	50648.1	301.9
172.mgrid	Multi-grid Solver: 3D Potential Field	95.5	175470.3	1066.1
173.applu	Parabolic/Elliptic Partial Differential Equations	99.2	121798.5	736.9
301.apsi	Meteorology: Pollutant Distribution	107.9	456479.1	2786.7

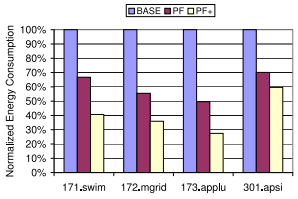


Figure 11: Normalized energy consumptions.

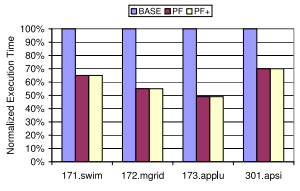


Figure 12: Normalized execution times.

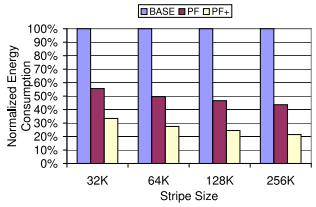


Figure 13: Normalized energy consumptions with the different stripe sizes.

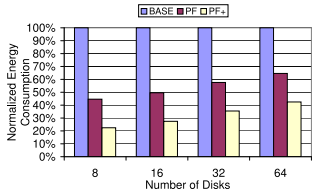


Figure 14: Normalized energy consumptions with the different stripe factors.

significant amount of disk energy savings with little impact on the performance improvement achieved by the PF scheme.

6.3 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 13 gives the normalized energy consumptions with the different stripe sizes (ranging from 32KB to 256KB). Recall from Table 1 that the default stripe size was 64KB. The values of the all other simulation parameters are fixed at the values given in Table 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 14 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 13 and 14 clearly show that our approach is successful across a range of values for stripe sizes and the number of disks.

7. CONCLUDING REMARKS AND FUTURE WORK

The main contribution of this paper is 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. To test the effectiveness of the proposed strategy, we implemented it within an optimizing compiler and made experiments with four applications that manipulate disk-resident data arrays. The results obtained so far from our experiments are very encouraging and show that the energy-aware prefetching brings significant energy benefits over a state-of-the-art (performance oriented) I/O prefetching scheme, without degrading the performance of the latter. Our ongoing work involves integrating this optimization with well-known I/O optimizations such as collective I/O [27] and caching. We are also planning to enhance our approach to accommodate disk spin-downs as well, in addition to multi-speed disks.

8. REFERENCES

[1] Omega library. `http://www.cs.umd.edu/projects/omega`.

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

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

[4] P. H. Carns, W. B. L. III, R. B. Boss, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of*

the 4th Annual Linux Showcase and Conference, pages 317–327, October 2000.

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

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

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

[8] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Survey*, 26(2):145–185, 1994.

[9] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, July 2002.

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

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

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

[13] M. Elnozahy, M. Kistler, and R. Rajamony. Energy Conservation Policies for Web Servers. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

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

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

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

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

[18] R. K. K. Li, P. Horton, and T. Anderson. A Quantitative Analysis of Disk Drive Power Management in Portable Computers. In *Proceedings of the USENIX Winter Conference*, pages 279–292, 1994.

[19] K. Okada, N. Kojima, and K. Yamashita. A novel drive architecture of HDD: multimode hard disc drive. In *Proceedings of the International Conference on Consumer Electronics*, pages 92–93, June 2000.

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

[21] E. Pinheiro and R. Bianchini. Energy Conservation Techniques for Disk Array-Based Servers. In *Proceedings of the 17th International Conference on Supercomputing*, pages 66–78, June 2004.

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

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

[24] S. W. Son, G. Chen, M. Kandemir, and A. Choudhary. Exposing Disk Layout to Compiler for Reducing Energy Consumption of Parallel Disk Based Systems. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 174–185, June 2005.

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

[26] SPEC. Specfp 2000. `http://www.specbench.org/cpu2000/CFP2000/`, 2000.

[27] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, 1999.

[28] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 30–44, New York, NY, USA, 1991. ACM Press.

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

[30] H. Yada, H. Ishioka, T. Yamakoshi, Y. Onuki, Y. Shimano, M. U. H. Kanno, and N. Hayashi. Head positioning servo and data channel for HDD's with multiple spindle speeds. *IEEE Transactions on Magnetics*, 36(5):2213–2215, September 2000.

[31] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: Helping disk arrays sleep through the winter. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[32] Q. Zhu, F. M. David, C. F. Devaraj, Z. Li, Y. Zhou, and P. Cao. Reducing Energy Consumption of Disk Storage Using Power-Aware Cache Management. In *10th International Conference on High-Performance Computer Architecture*, pages 118–129, 2004.

[33] Q. Zhu, A. Shankar, and Y. Zhou. PB-LRU: a self-tuning power aware storage cache replacement algorithm for conserving disk energy. In *Proceedings of the 18th Annual International Conference on Supercomputing*, pages 79–88, 2004.