Disk Layout Optimization for Reducing Energy Consumption*

S. W. Son G. Chen M. Kandemir
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802, USA
{sson,gchen,kandemir}@cse.psu.edu

ABSTRACT

Excessive power consumption is becoming a major barrier to extracting the maximum performance from high-performance parallel systems. Therefore, techniques oriented towards reducing power consumption of such systems are expected to become increasingly important in the future. Since disk systems of high-performance architectures are known to constitute a large fraction of the overall power budget, they form an important optimization target. Previous work on disk power management focuses primarily on hardware based schemes. However, since disk access pattern, i.e., the order in which disks on a system are accessed, is mainly shaped by the program code access pattern and disk layout of data, software techniques can also play a critical role in disk power management. Motivated by this observation, this paper proposes and evaluates a profile-driven disk layout optimization scheme for reducing energy consumption. The proposed scheme analyzes the array access traces obtained through profiling and determines, for each disk-resident data structure, the start disk from which the data is striped, the number of disks over which the data is striped, and the stripe unit. This paper discusses implementation details of our approach and presents an experimental evaluation of it. Our experiments with the entire suite of Spec95 floating-point benchmarks that are modified to operate on disk-resident data show that the proposed approach is very effective in reducing disk energy consumption. The results also show that the performance degradation caused by our approach is very small. This paper also compares our approach to a code restructuring based optimization mechanism and discusses how the two techniques can be combined for achieving the best results.

Categories and Subject Descriptors

B.4 [**Input/Output and Data Communications**]: Input/Output Devices; D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

General Terms

Algorithms, Design, Performance, Experimentation

Keywords

Optimizing Compiler, Disk Layout, Low Power

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

ICS'05, June 20–22, Boston, MA, USA.

1. INTRODUCTION

Reducing power consumption is becoming increasingly important for high-end cluster/server based systems. These systems are radically different from embedded systems where most of the prior power-related studies appeared. While in embedded systems power research is generally driven by desire of increasing battery lifetime, in high-end computing both economic and environmental factors play an important role. In particular, recent research [5, 13] indicates that a large portion of the system maintenance budget in high-end systems is invested in cooling due to excessive power consumption of such systems. In addition, high power consumption, which requires sophisticated power generation and transmission technologies, is known to be harmful to the environment. As a result, recent years have witnessed several efforts on reducing power consumption of high-end systems [8, 13, 7].

Disk systems of large, high-performance machines are known to contribute to a significant fraction of overall power budget. Motivated by this observation, recent studies such as [23] and [26] specifically focused on disk system and proposed energy saving strategies. These efforts are either hardware based (e.g., reducing disk speed if the associated latency can be tolerated by the application) or software based (e.g., restructuring code for taking best advantage of available low-power capabilities provided by the disk system). From the software angle, there are two major parameters that can be tuned for low power: the code structure of the application program and the disk layout of data. This paper focuses on the second parameter and proposes a disk layout detection scheme for reducing power consumption on the disk system. The importance of disk layout from the power angle stems from the fact that it determines the order in which the different disks in the system are accessed and their access durations.

Our approach saves disk power by increasing the number and length of disk idle periods. It is known from the prior research [16, 26] that the array-based scientific codes are not able to take much advantage of conventional, hardware-based disk low-power management, mainly due to short disk idle periods. Through fine-tuning of layouts of individual arrays, our approach is able to increase disk idle times substantially; and, this in turn enables effective disk power management. Specifically, focusing on array/loop-intensive scientific applications with regular data access patterns, this paper makes the following contributions:

- We present an algorithm for determining the disk layouts of array data. The goal of this profile-driven algorithm is to increase disk idleness and improve the effectiveness of the underlying disk power management mechanism supported by the hardware.
- We discuss details of our algorithm and present an experimental evaluation of it. Our experiments with the entire suite of Spec95 floating-point benchmarks that are modified to operate with disk-resident data show that the proposed approach is very effective in reducing disk energy consumption. Our results also show that the performance degradation caused by the proposed approach is very small.
- We discuss how our approach can be combined with code restructuring. The results with the benchmark codes in our experimental suite indicate that this combined approach, which applies layout optimization followed by code restructuring, generates better results than pure layout optimization and pure code structuring.

A unique characteristic of the work presented in this paper is that it automatically determines disk layouts for energy efficiency. To the best of our knowledge, this is the first study along this direction. Note that many file systems allow explicit tuning of disk layouts on an individual file basis and this property can be used to convey the layout information extracted by our approach to the file system.

The remainder of this paper is structured as follows. The next section discusses related work. Section 3 discusses the disk layout abstraction and the power management abstraction our approach employs. The details of our approach are presented in Section 4. Section 5 discusses how the proposed disk layout detection scheme can be combined with loop restructuring. Section 6 presents experimental evidence that demonstrates the effectiveness of the proposed approach and we conclude the paper in Section 7.

2. RELATED WORK

There have been numerous proposals targeting at energy/ power reduction in low-cost embedded systems [28, 15, 3, 4, 14, 30, 27, 22]. Since our work focuses on high-end systems, in this section, we mainly discuss the power related hardware and software efforts on these systems.

We can divide the related work on disk power management into two groups. In the first group are the hardware-based efforts. The studies presented in [10, 19, 11], which are designed for laptop disks, save disk power by spinning down a disk when it becomes idle during execution. Such a disk is reactivated (spun up) when an access is made to it. Since spinning up and spinning down a disk both take extra cycles and consume extra energy, one needs to be conservative in exercising this option. Specifically, if the idleness is not large, it may not be a good idea to spin down a disk. Therefore, accurate prediction of idleness may be an important issue for this mechanism to be successful. Observing that spinning down an idle disk may not be very effective in server workloads, [16] proposed a different approach which is based on running the disk with a reduced speed. Since such a disk can still serve requests, this approach can potentially utilize even small idle periods. In fact, [16] shows both performance and energy benefits of this approach over the spinning-down based approach. [6] presents an approach that demonstrates how a disk system constructed using disks with different speeds can be utilized effectively for reducing energy consumption.

On the software front, [26] shows how an application code can be restructured for reducing disk power. They also show that code restructuring can be used in conjunction with both spinning down disks and in an architecture that supports disks with different speeds. As compared to the hardware-based work, our approach takes a totally different stand. While we focus on a particular application domain (namely array/loop-intensive scientific codes that perform I/O), we allow the software to control disk layout, and in this way, we can optimize codes that are not amenable to hardware-only approaches (due to short disk idle times). In a sense, the work presented here and [26] are complementary. In fact, our experimental evaluation indicates that the best energy behavior is obtained by employing both code restructuring and data layout optimization. Apart from disk power, there exist several studies that target cluster systems as a whole [8, 7, 12, 13] or interconnection networks in particular [9, 20]. Since we focus exclusively on disk system, our approach can be used with many of these power-saving techniques proposed in the past.

3. ABSTRACTIONS USED BY OUR APPROACH

There are two important types of information that needs to be fed to our approach, for it to optimize disk layouts for saving energy. The first one is the disk layout abstraction, which captures the parameters whose values can be tuned by our approach. The second one is the energy management technique/strategy supported by the underlying hardware. In the following two subsections, we discuss these in more detail.

3.1 Disk Layout Abstraction

In this section, we describe the disk layout abstraction used by our 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 [21]. 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 pointed out below. In this work, we represent disk layout of an array using a triplet of the form:

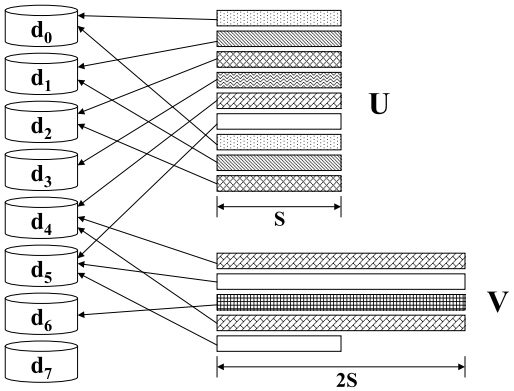


Figure 1: Two different example disk layouts. Left: $(d_0, 6, S)$ **and Right:** $(d_4, 3, 2S)$**.**

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

The first component, start_disk, in this triplet indicates the disk from which the array is started to get striped. The second component, stripe_factor, gives the number of disks used to stripe the data, and the third component, stripe_size, gives the stripe (unit) size used. 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 [25], 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. Two example disk layouts for two-dimensional disk-resident arrays are depicted in Figure 1. The first layout (i.e., the one for array U) is $(d_0, 6, S)$, whereas the second layout (i.e., the one for array V) is $(d_4, 3, 2S)$.

Since a triplet is used for representing disk layout in our work, our job is to determine the three layout parameters for each disk-resident array that needs to be created by a given application program. It needs to be noted however that this has to be done in a coordinated fashion by considering all the disk-resident arrays in the application. This is because the different disk-resident arrays can potentially share the same set of disks and determining their layouts in an independent fashion can lead to unpredictable results (e.g., due to irregular disk access patterns) at runtime as far as saving disk power is concerned. In this paper, we present an algorithm embedded within a trace analyzer for determining disk layouts of array data to minimize energy consumption, under the assumption that each array is stored in a separate file.[1]

3.2 Power Management Abstraction

Figure 2 depicts the transitions between the different states supported by the disks assumed in this study. The labels attached to the arcs in this figure indicate how the transitions are triggered. Basically, we assume that each disk is equipped with a timer-based power management capability. In this mechanism, when the current access to a disk is completed, the disk transitions to the idle state. If it remains in the idle state for a certain amount of time, it is spun down. We say in this case that the disk is placed into the *low-power operation mode.* The disk transitions back to the active mode by spinning up when a new request to it is made. Note that this model represents one of the simplest mechanisms that can be supported by

[1]We can relax this constraint by allowing one-to-many or many-to-one mappings between the arrays and the files, as will be discussed later.

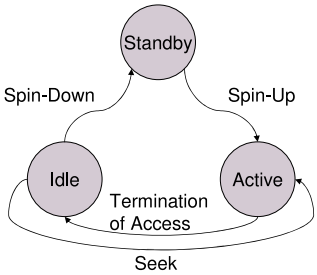


Figure 2: Different disk states and transitions among them. When computing energy consumption, we take into account the energies consumed at each state.

a server disk that allows power management. The details of the parameters used in this model will be given later when we present our experimental platform. For now, the important point to emphasize here is that, since spinning-down and spinning-up take both extra time and energy, they need to be minimized. Therefore, one would prefer, from both performance and power consumption angles, a few long idle periods over numerous short idle periods. The goal of the profile-driven approach proposed in this paper is to increase the duration of idle periods, thereby increasing the chances for this power management scheme to be applicable and successful. The proposed approach achieves this by setting the layout parameters for each disk-resident array manipulated by the application code. We also need to mention that there exist more effective power management mechanisms recently proposed in the literature. Though we experimented our approach under different mechanisms, called DRPM [16], we do not present the results in this paper due to lack of space.

4. DISK LAYOUT DETECTION ALGORITHM

4.1 High-Level View

As stated earlier, the main goal behind our approach is to determine a suitable disk layout for each disk-resident array manipulated by the application so that the energy spent on the disk system during execution is minimized. Our approach reduces energy consumption by increasing disk idle periods. However, considering only energy consumption alone may not be a wise choice since performance of the application is also important and affected significantly by the disk layouts chosen for its arrays. After all, if there was no performance concern, one could potentially work with a single disk, thereby placing all the remaining disks in the system into the low-power operation mode. However, such an approach would hardly be acceptable from the performance perspective. Therefore, our objective is to strike a balance between energy consumption and performance; that is, we would like to save as much energy as possible without significantly impacting original execution cycles (i.e., execution cycles that would be taken by a pure performance-oriented approach that does not employ any power-saving strategy).

An important property of our algorithm is that it is profile driven (as shown in Figure 3). The application code is first instrumented and then profiled using a typical input set. The inserted instrumentation code records information on each disk access issued by the application program. At the end of this profiling, we obtain an *array access sequence*, which is of the form $(A_1, A_2, ..., A_n)$. Each array access A_i in this sequence has the form (X, a, t), where X is the id of the disk-resident array, a is the offset of the accessed array element within the array, and t is the time stamp. The time stamp of an array access is the time since the start of the program after deducting the I/O time spent in the disk accesses. As a simple example, let us assume a disk access is issued 300ms after the program starts its execution, and during the first 300ms of the program execution, disk I/O takes 100ms in total. Then the time stamp for this array access is calculated as 300ms − 100ms = 200ms. We say that two array accesses $A_i = (X, a_1, t_1)$ and $A_j = (Y, a_2, t_2)$ *conflict* with each other if they access the same disk and the difference between their time stamps is less than the disk response time (denoted by R). Further, we call the conflicts due to accessing the same array (i.e., $X = Y$) the *intra-array conflicts*, and the conflicts due to accessing different arrays (i.e., $X \neq Y$) are referred to as the *inter-array conflicts*. Note that, if two array accesses conflict with each other, one of them has to be delayed, which causes the program execution to slow down. Based on this discussion, we can re-state the goal of our approach as one of reducing energy consumption on the disk system while minimizing the number of intra-array and inter-array conflicts as much as possible.

Figure 3: The connection between profiling and layout optimization.

Our approach determines the three components of the disk layout of each array in the application: stripe factor, stripe size, and start disk. It needs to be noted that these three parameters are actually inter-related. What this means is that they affect one another and selecting a value for one of them restricts the potential search space for the other two parameters. Ideally, one would want an algorithm that would try all potential values for all these three parameters of the layout and select the one that generates the best trade-off between energy consumption and performance. However, such an approach is not feasible in general. This is mainly because the search space of potential solutions can be very large. First, in a system with large number of disks, we have a lot of candidates for the start disk and stripe factor for a given array. Second, one can have many choices for the stripe size, depending on the capabilities of the underlying file system. Third, in order to reach optimal results, one needs to try all potential layout combinations for all disk-resident arrays. All these factors make an exhaustive search infeasible in practice except for cases with a few disks and a few arrays. Consequently, our approach is essentially a fast heuristic that generates not optimal but close-to-optimal results for most cases.

The proposed approach determines a single component of a disk layout at a time. More specifically, it first determines the stripe factor for all arrays and then the stripe size for all arrays. Then, based on these, it determines the start disk for all arrays. The reason that we first determine the stripe factor and the stripe size is that, these two parameters affect the magnitude of the intra-array conflicts. Once we determine the stripe factor and the stripe size for each array independently, the part of the algorithm that determines the start disk for arrays is executed. This part positions the arrays on the disk system in such a fashion that the number of inter-array conflicts is minimized as much as possible. In the next section, we present the technical details of our layout optimization approach.

4.2 Details of the Algorithm

4.2.1 Determining Stripe Factor

Figure 4 gives our algorithm for determining the stripe factor for each disk-resident array. Generally speaking, storing an array in more disks can reduce the number of intra-array conflicts, which in turn can improve overall performance. However, storing an array in more disks also means that more disks need to be activated (i.e., they cannot be placed into the low-power mode). This clearly increases energy consumption on the disk system. Therefore, the goal of our algorithm given in Figure 4 is to minimize the stripe factor for each array while keeping the number of intra-array conflicts within a tolerable range. Our algorithm maintains a queue (denoted by $Q[X]$) for each array X, which contains the most recent accesses to array X that may create intra-array conflicts. For each array X, the algorithm uses an array of counters ($K[X][1..D]$) to capture the distribution of the length of $Q[X]$. Specifically, array K records the fact that, throughout the execution of the program, we observed $K[X][i]$ times $Q[X]$ contains exactly i accesses to array X, that is, i accesses to array X conflict with each other. The value of $\sum_{i=1}^{d} K[X][i]$ indicates the number of accesses to array X that cannot be served without intra-array conflicts if X is stored in less than d disks (i.e., if its stripe factor is less than d). Storing X in d disks can eliminate these conflicts. The stripe factor (F_X) for array X is the minimum integer value that satisfies the following constraint:

$$\frac{\sum_{i=1}^{F[X]} K[X][i]}{\sum_{i=1}^{D} K[X][i]} \geq T,$$

Input: trace file
Output: stripe factor for each array

D — the number of disks;
R — the disk response time;
T — the threshold, $0 < T \leq 1$;
$Q[X]$ — the access queue for array X;
$K[X][1..D]$ — the counters for array X;
$F[X]$ — the stripe factor for array X;

while(there are array accesses to be processed) {
  assume the current array access is (X, a, t);
  $U = \{(X, a', t') \mid (X, a', t' \in Q \text{ and } t - t' > R\}$;
  $Q[X] = (Q[X] - U) \cup \{(X, a, t)\}$;
  $i = |Q[X]|$;
  if($i > D$) $i = D$;
  $K[X][i] = K[X][i] + 1$;
}
for each array X {
  // determine the stripe factor for array X.
  $sum = \sum_{i=1}^{D} K[X][i]$;
  $p = 0$; $F[X] = 0$;
  while($p < T$) {
    $F[X] = F[X] + 1$;
    $p = p + K[X][F[X]]/sum$;
  }
}

Figure 4: The algorithm for determining the stripe factor for each array.

where threshold T ($0 < T \leq 1$) determines the percentage of intra-array conflicts we want to eliminate. It is to be noted that T is a parameter whose value can be tuned by the programmer. This allows us perform a trade-off analysis between energy and performance. In our experimental evaluation, we conduct a sensitivity analysis regarding parameter T.

4.2.2 Determining Stripe Size

Figure 5 gives our algorithm for determining the stripe size for each disk-resident array. When the stripe factor has been determined, whether two array elements of an array X are located in the same disk or not is determined by the stripe size of X. Consequently, the stripe size determines the number of intra-array conflicts. Our algorithm computes the number of conflicts for each array with each available stripe size (which is given as an input to the algorithm), and selects the stripe size with the minimum number of intra-array conflicts.

4.2.3 Determining Start Disk

Figure 6 gives the algorithm used to determine the start disk for each array. The goal of this algorithm is to minimize the total number of inter-array conflicts. In this algorithm, each array is divided into a set of sub-arrays such that the elements that are stored on the same disk belong to the same sub-array. Obviously, array X with stripe factor of $F[X]$ is divided into $F[X]$ sub-arrays. Further, given the element size s, the stripe factor $F[X]$, and the stripe size $S[X]$ of array X, the i^{th} sub-array of X can be calculated as:

$$\{X[a] \mid \lfloor a \times s/S[X] \rfloor \bmod F[X] = i\}.$$

Our algorithm operates in two steps. In the first step, based on the profile data collected, we compute the number of potential inter-array conflicts between each pair of sub-arrays, that is, the number of inter-array conflicts due to accessing each pair of sub-arrays if these two sub-arrays are stored on the same disk. In the second step of the algorithm, we determine the start disk for each array. Instead of exhaustively searching the entire solution space for the optimal result, we determine the start disk for each array using a greedy search based strategy. In Figure 6, we can see that the main body of this step is a loop. At each iteration of this loop, we determine the start disk for a single array. Specifically, assuming the set V contains the arrays whose start disks have already been determined, we select d_X ($0 \leq d_X < D$) as the start disk node for the next array $X \notin V$ such that the value of the following expression is minimized:

$$\sum_{Y \in V} \text{conflict}(X, d_X, Y, d_Y), \quad (1)$$

Input: trace file and stripe factor for each array
Output: stripe size for each array

R — the disk response time;
N — the number of available stripe sizes;
$Q[X]$ — the access queue for array X;
$Z[1..N]$ — the available stripe sizes;
$C[X][1..N]$ — the conflict counters for array X;
$F[X]$ — the stripe factor for array X;
$S[X]$ — the stripe size for array X;

while(there are array accesses to be processed) {
  assume the current array access is (X, a, t);
  $U = \{(X, a', t') \mid (X, a', t' \in Q \text{ and } t - t' > R\}$;
  $Q[X] = Q[X] - U$;
  for $i = 1$ to N {
    $d = \lfloor a/Z[i] \rfloor \bmod F[X]$;
    for each $(X, a', t') \in Q[X]$ {
      if($\lfloor a'/Z[i] \rfloor \bmod F[X] = d$)
        $C[X][i] = C[X][i] + 1$;
  }
  $Q[X] = Q[X] \cup \{(X, a, t)\}$;
}
for each array X {
  // determine the stripe size for array X.
  $min = \infty$;
  for $i = 1$ to N
    if($C[X][i] < min$) {
      $S[X] = Z[i]$; $min = C[X][i]$;
    }
}

Figure 5: The algorithm for determining the stripe size for each array.

where the function conflict(X, d_X, Y, d_Y) computes the number of inter-array conflicts between array X and Y whose start disks are d_X and d_Y, respectively. Note that, for any given $Y \in V$, d_Y has been determined in the previous iterations of the loop. The implementation of function conflict(X, d_X, Y, d_Y) is also given in Figure 6. If multiple d_X values yield the same value of Expression (1), we prefer the minimum value of d_X. This is to reduce the number of disks used by the program, and thus reduce the overall energy consumption of the system.

In our implementation, these three algorithms, namely, the algorithms given in Figures 4, 5, and 6, form the trace analyzer (see Figure 3).

4.3 Discussion of Array-to-File Mapping

So far we have assumed that each disk-resident array is stored in a single file, and each file stores a single disk-resident array (i.e., 1-to-1 mapping between files and arrays). However, it is possible in some environment that, a large array may be split into multiple parts (sub-arrays) and stored in multiple files. For example, array $X[0..m][0..n]$ in the following code fragment can be stored in m files (F_1, F_2, ..., F_m) in such a fashion that file F_i stores $X[i][0..n]$:

```
for i = 0 to m {
  for j = 0 to n {
    ...X[i][j]...
  }
}
```

Since the files storing the elements of array X are accessed using the same piece of array access code, these files need to have the same disk layout in most cases. Our scheme can be extended to address this situation. Specifically, we can allow the programmer to specify array-to-file mapping for each disk-resident array and the constraints on which files should have the same disk layout. In practice, array file mapping can be specified using a function f:

$$f(X, \vec{I}) = F_i,$$

where X is the name of the array, $\vec{I}$ is the subscript vector, and F_i is the name of the file. We refer to the set of files that must have an identical disk layout as a *file group*. The constraints on file grouping can be expressed using a function g:

$$g(F_i) = G_j,$$

```
Input: trace file, stripe factor, and stripe size for each array
Output: start disk for each array

D — the number of disks;
R — the disk response time;
Q — the access queue;
F[X] — the stripe factor for array X;
S[X] — the stripe size for array X;
C[X][i][X'][j] — the number of inter-array conflicts between the
                 i^th sub-array of X and the j^th sub-array of X'
                 if these two sub-arrays are stored on the same disk.
W[X] — the start disk for array X;

//step 1: computing C[X][d][X'][d']
while(there are array accesses to be processed) {
  assume the current array access is (X, a, t);
  d = ⌊a/S[X]⌋ mod F[X];
  U = {(X', a', t') | (X', a', t' ∈ Q and t − t' > R};
  Q = Q − U;
  for each (X', a', t') ∈ Q {
    if(X' ≠ X) {
      d' = ⌊a'/S[X']⌋ mod F[X'];
      C[X][d][X'][d'] = C[X][d][X'][d'] + 1;
      C[X'][d'][X][d] = C[X'][d'][X][d] + 1;
    }
  }
}
// step 2: determining the start disk for each array
V = φ;
for each array X {
  min = ∞;
  for i = 0 to D − 1 {
    c = 0;
    for each Y ∈ V
      c = c + conflict(X, i, Y, W[Y]);
    if(c < min) {
      W[X] = i; min = c;
    }
  }
  V = V ∪ {X};
}

// auxiliary function: computes the number inter-array conflicts between
// arrays X and Y whose start disks are P_X and P_Y, respectively.
int conflict(X, P_X, Y, P_Y) {
  for i = 0 to D − 1 { B[i] = −1; }
  for i = 0 to F[X] − 1
    B[(i + P_X) mod D] = i;
  c = 0;
  for i = 0 to F[Y] − 1 {
    j = B[(i + P_Y) mod D];
    if(j ≠ −1)
      c = c + C[X][j][Y][i];
  }
  return c;
}
```

Figure 6: The algorithm for determining the start disk for each array.

where F_i is a file and it belongs to the file group G_j. For ease of discussion, we define a *group element*, $G_j[k]$, as follows:

$$G_j[k] = \{X[\vec{I}] \mid f(X, \vec{I}) \in G_j \wedge h(X, \vec{I}) = k\},$$

where k is an integer, and $h(X, \vec{I})$ is the offset of $X[\vec{I}]$ within file $f(X, \vec{I})$.

Our scheme can be extended to determine the disk layouts that satisfy the constraints on the grouping by modifying the instrumentation tool such that, for each array access in the trace file, each array element is replaced by the group element that contains this array element. Consequently, the files that belong to the same file group are considered as a single *abstract file*. Our scheme then determines the disk layout for the abstract file of each file group. This layout is then applied to all the files that belong to this file group.

4.4 Example

In this subsection, we illustrate how our approach works in practice by applying it to an example code fragment. Figure 7(a) presents an example code fragment. In this example, we set the value of the T parameter to 1, and we use a single file per disk-resident array (1-to-1 mapping). For illustration purposes, let us assume that the underlying I/O system has 6 disks, the response time for a disk is 5ms, and each loop iteration (of loops L1, L2, and L3 shown in the figure) takes 10ms. By analyzing the disk access trace of this program, our approach determines that array X must be stored in at least two disks and that array Z must be stored in at least three disks. If we stored X in a single disk, the two accesses to X in each iteration of loop L1 would conflict with each other. Specifically, the access to $X[i + 1024]$ had to wait for the access to $X[i]$ to complete. Similarly, if array Z were stored in fewer than three disks, the three accesses to Z in each iteration of loop L2 would conflict with one another. Our approach determines the stripe factors for arrays X, Y, and Z as 2, 1, and 3, respectively, since we want to minimize the number of disks used for storing each array without increasing the number of intra-array conflicts (so that we can save energy without impacting performance too much).

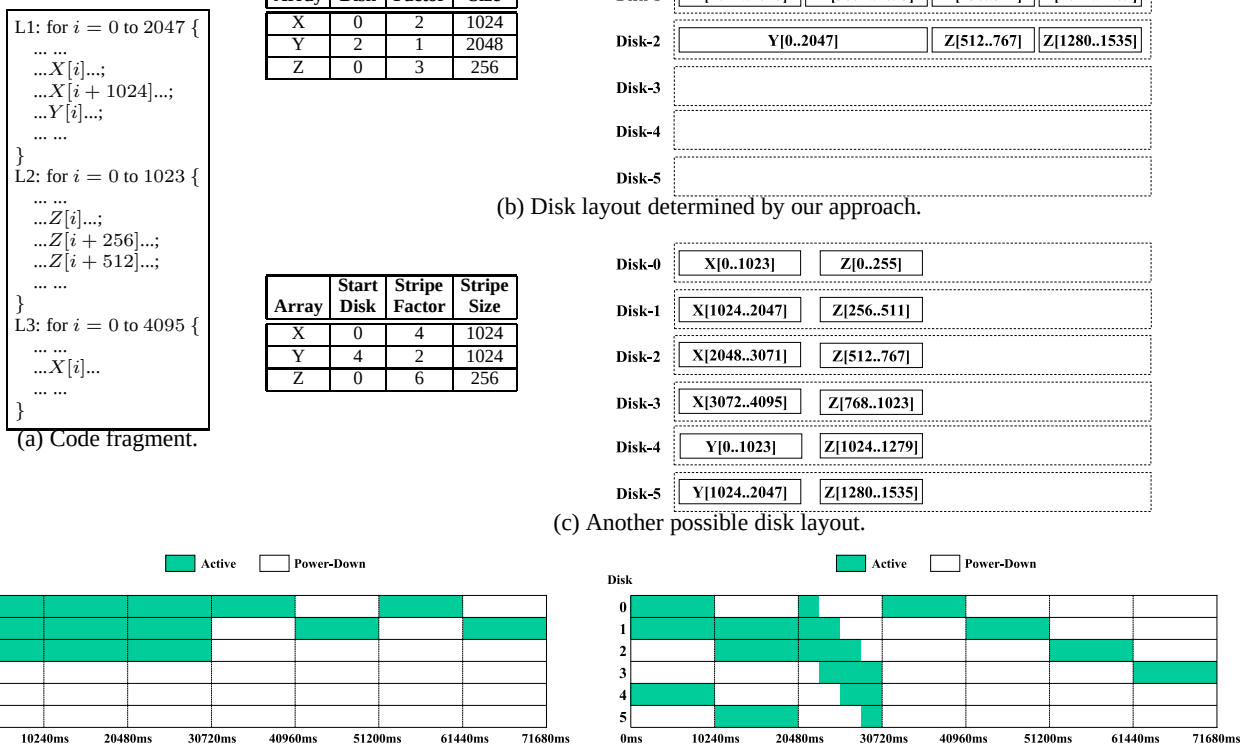
We determine stripe size for each array once the stripe factor for that array is determined. Our trace analyzer analyzes the trace and calculates the number of intra-array conflicts for each array with each possible stripe size. Let us assume, again for illustration purposes, that the underlying disk architecture and file system support four stripe sizes: 256B, 512B, 1024B, and 2048B. For array X in loop L1, the number of intra-array conflicts with the stripe sizes 256B, 512B, 1024B, and 2048B, are 2048, 2048, 0, and 1024, respectively. Our algorithm selects the stripe size with the minimum number of intra-array conflicts (1024B). We do not need to further consider the stripe size for array Y since its stripe factor is one, i.e., Y is stored in only a single disk. The number of intra-array conflicts for array Z with stripe sizes 256B, 512B, 1024B, and 2048B, are 0, 1024, 2048, and 3072, respectively. Therefore, we select 256B as the stripe size for array Z.

To determine the start disk for each array, our algorithm counts the number of inter-array conflicts between each pair of disk stripes. It then determines the start disk for each array, one array after another. In our example code fragment, we have three arrays: X, Y, and Z. We first determine the start disk for array X. This step is trivial since we can pick any disk, say disk 0, as the start disk for array X. And then, we determine the start disk for array Y. At this step, we try all possible start disks for Y, and select one that minimizes the inter-array conflicts between X and Y. Finally, we determine the start disk for array Z. At this step, we need to select the start disk for array Z such that the total number of inter-array conflicts between X and Z and that between Y and Z are minimized. Figure 7(b) gives the final disk layouts determined by our approach for this example, while Figure 7(c) gives another possible disk layout. It is to be emphasized that both these layouts have the same disk conflicts. However, by comparing disk power states[2] presented in Figure 7(d) and 7(e), we observe that the disk layouts determined by our approach exhibit much better idle periods. Specifically, we see that our algorithm uses three disks to store all the arrays used in the program so that the other three disks in the system can remain in the low-power mode throughout the entire execution, and this can lead to significant energy savings at runtime. Further, in Figure 7(d), we observe a total of six reactivations (i.e., switching a disk from the low-power mode to the active mode), whereas in Figure 7(e), we have a total of thirteen reactivations. Note that reactivating a disk incurs both performance and energy penalties. This small example clearly demonstrates that our approach increases opportunities for saving disk energy.

5. COMBINING LAYOUT OPTIMIZATION WITH LOOP RESTRUCTURING

Our approach determines the disk layouts of the arrays in a given program such that the number of disk conflicts and the number of disks used to store arrays are minimized. However, this does not mean that all components of a given application will work well under the disk layouts determined by our profile-driven approach. In particular, while it is expected that the layouts found by our approach will operate well for most of the loop nests of the application, it is still possible that these layouts perform poorly for a couple of nests. This can certainly be the case for large applications with

[2] A power state diagram shows the states of the disks over time.



(a) Code fragment.

(b) Disk layout determined by our approach.

(c) Another possible disk layout.

(d) Disk power states for (b).

(e) Disk power states for (c).

Figure 7: An example that illustrates how our trace analyzer works in practice.

Input: a loop nest and disk layouts for all arrays
Output: transformed (restructured) loop nest

$A[i]$ — the set of array elements stored in disk$_i$;
$S[\mathcal{L}]$ — the set of array elements accessed within $\mathcal{L}$;
$IT[\mathcal{L}]$ — the set of iterations of $\mathcal{L}$;

for each loop nest $\mathcal{L}$ of program $\mathcal{P}$ {
  $D = \phi$;
  for each disk$_i$ of the system {
    $T = A_i \cap S[L]$;
    $N_i = \{\vec{I} \mid \vec{I} \in IT[\mathcal{L}] \wedge \exists x \in T : x$ is accessed at $\vec{I}\}$;
    if($N_i \neq \phi$)
      $D = D \cup \{i\}$;
  }
  for each $K \subseteq D$ {
    $d = IT[\mathcal{L}]$;
    for each $i \in K$
      $d = d \cap N_i$;
    for each $i \in D - K$
      $d = d \cap (I[L] - N_i)$;
    output "turn off disks not in K";
    output "for $\vec{I} \in d$ {body of $\mathcal{L}$}"
  }
}

Figure 8: Our loop restructuring algorithm.

many loop nests. There are two potential solutions to this problem: (1) allowing such nests to operate with different disk layouts, or (2) using code restructuring to change the disk access patterns of these nests. Unfortunately, the first option does not sound very realistic. It is important to note that, once the disk layouts are assigned and arrays are created, it is difficult to change the layouts. This is because such a change would normally require costly data remappings on the disk system, whose overheads may not be tolerable in practice. In the rest of this section, we explore the second approach, namely, changing the data access patterns through code restructuring (loop transformation). The proposed approach is applied to each loop nest whose behavior is not good under the disk layouts determined by our trace analyzer.

We assume that a loop nest $\mathcal{L}$ (whose access pattern we want to modify through loop restructuring) can be represented as follows:

$$\text{for } \vec{I} = \vec{L} \text{ to } \vec{U} \ \{\text{body}\}$$

where $\vec{I}$ is the iteration vector (containing the loop iterators from top to bottom); and $\vec{L}$ and $\vec{U}$ are the vectors that hold the lower and upper bounds for the loops, respectively. We denote the set of the disks used in loop iteration $\vec{I}$ as $d(\vec{I})$. The goal behind our loop restructuring approach is to *cluster*, at a given time, disk accesses to as fewer disks as possible. Note that, this code restructuring approach works with the layout information determined by our trace analyzer.

We start by making the following definitions:

$$D(\mathcal{L}) = \{d(\vec{I}) \mid \vec{L} \preceq \vec{I} \preceq \vec{U}\} = \{d_1, d_2, ..., d_n\};$$
$$S(\mathcal{L}, d_i) = \{\vec{I} \mid d(\vec{I}) = d_i\}.$$

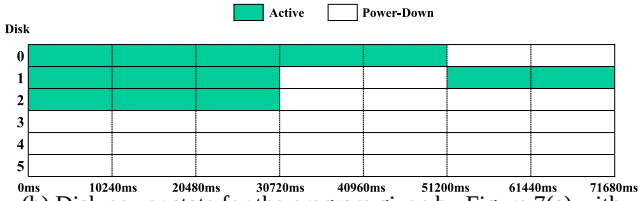
Here, each d_i indicates a subset of disks that are accessed by at least one iteration. Consequently, we can restructure loop nest $\mathcal{L}$ as:

$$\text{for } \vec{I} \in S(\mathcal{L}, d_1) \ \{\text{body}\}$$
$$\text{for } \vec{I} \in S(\mathcal{L}, d_2) \ \{\text{body}\}$$
$$.....$$
$$\text{for } \vec{I} \in S(\mathcal{L}, d_n) \ \{\text{body}\}$$

For the iterations in $S(\mathcal{L}, d_i)$, the disks that do not belong to d_i can be placed into the low-power mode to conserve energy. It should be noted however that, when restructuring a loop nest, all data dependences must be preserved. Specifically, a loop nest cannot be restructured using a particular transformation if this transformation

```
L3-1: for ii = 0 to 1 {
  for i = ii * 2048 to ii * 2048 + 1023 {
    ... ...
    ...X[i]...
    ... ...
  }
}
L3-1: for ii = 0 to 1 {
  for i = ii * 2048 + 1024 to ii * 2048 + 2047 {
    ... ...
    ...X[i]...
    ... ...
  }
}
```

(a) Restructured loop L3 in the code fragment in Figure 7(a).



(b) Disk power state for the program given by Figure 7(a) with loop nest restructuring, based on disk layout given in Figure 7(b).

Figure 9: An example that illustrates the working of our loop restructuring algorithm.

might violate a data dependency within this loop nest. Further, since disk layouts have already been determined, given a d_i, we can compute $S(\mathcal{L}, d_i)$ at compilation time. In this paper, we consider only the loop nests where $S(\mathcal{L}, d_i)$ can be represented using Presburger formulations [24].[3] Therefore, a loop nest of the form

$$\text{for } \vec{I} \in S(\mathcal{L}, d_i) \text{ \{body\}}$$

can be built using a tool such as the Omega Library [1] at compile time. That is, using the Omega library (or a similar polyhedral tool), we can identify the set of loop iterations that access a particular set of disks. In addition, we can increase the duration of the periods spent in the low-power mode by minimizing the number of disks whose states need to be changed when we transition from the current loop nest to the next one (considering the multiple loop nests created from the original loop nest to be restructured). This can be achieved by sorting the elements of $D(\mathcal{L})$ such that the value of the following expression is minimized:

$$\sum_{i=1}^{|D(\mathcal{L})|-1} |d_i \cup d_{i+1} - d_i \cap d_{i+1}|.$$

Note that, $d_i \cap d_{i+1}$ gives the set of disks that are used by the two consecutive loop nests, and $|d_i \cup d_{i+1} - d_i \cap d_{i+1}|$ gives the number of disks whose states must be changed when we transition from the i^{th} loop nest to the $(i+i)^{\text{th}}$. Our code restructuring algorithm is given in Figure 8.

As an example, based on the disk layouts given in Figure 7(b), the loop L3 given in Figure 7(a) can be restructured as shown in Figure 9(a). Figure 9(b) gives the disk power states for this restructured program. At this point, by comparing Figure 9(b) with Figure 7(d), one can see that loop nest restructuring reduces the number of reactivations from six to four. In addition, the two short power-down periods for the second disk are now combined into a single, longer period. Notice that, a longer idle period can allow us use the low-power mode which would not be possible with a shorter idle period.

[3]Presburger formulation is a class of logical formulas which can be built from affine constraints over integer variables, the logical connectives ($\vee$, $\wedge$, and $\neg$), and the existential and universal quantifiers ($\exists$ and $\forall$). In this work, we employ the Omega Library to manipulate integer tuple relations and sets, which are described using Presburger formulas.

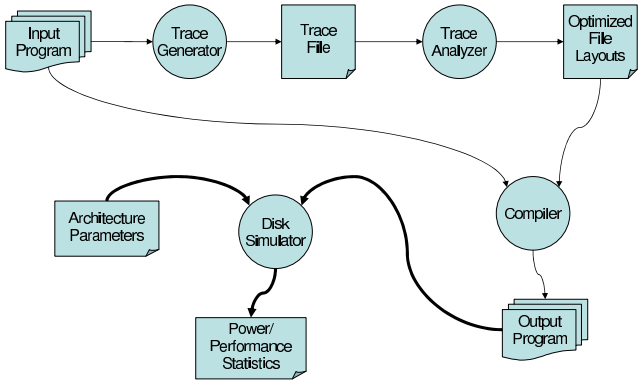


Figure 10: Implementation and experimental platform.

6. EXPERIMENTAL EVALUATION

In this section, we first present our experimental platform and methodology (Section 6.1), and then we give our experimental results (Section 6.2).

6.1 Implementation and Simulation Platform

Figure 10 shows our implementation and simulation platform. Let us first focus on the implementation. If we use only layout optimization (i.e., without code restructuring), we follow the following path. The (instrumented) input code is fed to a trace generator which generates a trace file that contains array accesses, as explained earlier in the paper. This file is then given to our trace analyzer, which determines the disk layouts for all disk-resident arrays manipulated by the application. The three specific components of our trace analyzer are described in Section 4. This layout information is then passed to the compiler along with the original code. The compiler (built upon SUIF [17]) modifies the application code to specify the layout of each array (this is typically done by inserting appropriate parameters in the file creation calls). If, on the other hand, we use disk layout optimization in conjunction with loop restructuring (explained in Section 5), the compiler also modifies the loop nests whose behaviors under the determined disk layouts are not satisfactory.

Let us now discuss the simulation environment, which is shown on the lower-left portion of Figure 10. We wrote a disk energy simulator to perform our experiments and gather power/performance statistics. In addition to the trace file, this disk simulator needs a model for the target disk system. Using these parameters, the simulator determines, for each request, the disks that need to be accessed and the duration of access for each disk. The default simulation parameters used in our experiments are given in Table 1. Both performance and energy statistics were calculated based on the figures extracted from the data sheet of the IBM Ultrastar 36Z15 [18], and are given in Table 1. The values for power mode transitions (see Figure 2) are also included in Table 1. In the rest of the paper, when we say "energy" we mean the energy consumed in the disk system. When we say "execution time/cycles", we mean the time/cycles it takes to complete the application execution. The disk energy consumption includes the energy consumptions during both active and idle periods, taking into account all the states that the disks go through during the entire execution (see Figure 2).

We performed experiments with five different schemes, which can be summarized as follows:

- *BASE:* This is the base version that does *not* use any energy saving strategy. This also represents the best execution time across all the schemes tested since it does not incur any performance penalty due to power management. It uses a fixed stripe size of 64KB for all arrays, and stripes all arrays over all available disks in the system, starting with the first disk. All the energy and performance (execution cycles) results presented for the remaining schemes are given with respect to this base version.
- *HW:* This is a pure hardware based conventional disk power management scheme used in studies such as [10] and [11]. It uses the same layouts as the *BASE* scheme. In this approach, however, 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, this ap-

Table 1: Default simulation parameters.

Parameter	Value
Disk Model	IBM Ultrastar 36Z15
Interface	SCSI
Storage Capacity	18 GB
Rotation Speed	15,000 RPM
Maximum Stripe Factor	8
Stripe Size Granularity	4 (16KB, 32KB, 64KB, 128KB)
Average seek time	3.4 msec
Average rotation time	2 msec
Internal transfer rate	55 MB/sec
Power (active)	13.5 W
Power (idle)	10.2 W
Power (standby)	2.5 W
Energy (spin down: idle → standby)	13 J
Time (spin down: idle → standby)	1.5 sec
Energy (spin up: standby → active)	135 J
Time (spin up: standby → active)	10.9 sec
T (in the stripe factor algorithm)	0.7

proach can incur significant performance degradations. Also, in order for this scheme to save power, the disk idleness should be large enough to compensate for the spin-up and spin-down latencies.

- *OPT:* This is the profile-driven disk layout detection scheme proposed in this paper. This scheme determines, for each disk-resident array manipulated by the application, the start disk, stripe size, and stripe factor. We also implemented three (more restricted) variants of this optimized scheme. In *OPT-1*, we force each array to get striped from the first disk (d_0) on the disk system, and determine the stripe size and stripe factor under this constraint. In *OPT-2*, we fix the stripe size at 64KB for all arrays, and determine the start disk and stripe factor. Finally, in *OPT-3*, we fix stripe factor at 8 (the total number of disks in the system) for all arrays, and determine the start disk and the stripe unit. Our main goal in making experiments with these three variants (*OPT-1*, *OPT-2*, and *OPT-3*) is to quantify the influence/importance of each of the three components of a disk layout individually.
- *LOOP:* This scheme uses the same fixed layout for each array as in the case of the *BASE* and *HW* schemes. However, it restructures code to increase disk idle periods. The specific transformation used is loop distribution, also known as loop fission [29]. This transformation places the statements in a given loop into separate loops, each with its own iteration space. One can expect this transformation to be useful from a disk energy viewpoint, in particular, in cases where it separates the references to different arrays, thereby minimizing the number of disks that need to be activated for a given loop.
- *LOOP+OPT:* This is the combined scheme discussed in Section 5. It first determines the disk layouts using *OPT*. After that, it identifies the loop nests that need to restructured so that they can be transformed to work well with the determined layouts. The purpose behind making experiments with this version is to check whether the combined approach brings any additional benefits over the *OPT* scheme.

Note that, except for the BASE scheme, all the versions work under the assumption that the disk hardware provides conventional disk power management capabilities (see Figure 2). The power consumption at each state and power/latency values in state transitions are given in Table 1 along with the other simulation parameters. The necessary code modifications required by the different schemes tested in this paper are automated within the SUIF infrastructure [17]. SUIF consists of a small kernel and a toolkit of compiler passes built on top of the kernel. The kernel defines an intermediate representation, provides functions to access and manipulate the intermediate representation, and structures the interface between compiler passes. In our experiments, the largest increase in compilation time occurred with the *LOOP+OPT* scheme, and was less than 40% (of the original compilation times) for all the benchmark codes tested.

Table 2 lists the benchmark codes used in our experimental evaluation. We used all 10 applications from the Spec95 floating-point benchmark suite [2]. In a pre-processing step, we made these benchmarks I/O intensive by making the arrays they manipulate disk resident. Each array is stored in a separate file on the disk system, and each file stores only a single array (i.e., 1-to-1 mapping). The

Table 2: Our benchmarks and their characteristics.

Benchmark	Brief Description	Data Size (GB)	Energy (J)	Time (sec)
101.tomcatv	Vectorized mesh gen	39.9	4817.1	96.2
102.swim	Shallow water eqn	55.6	6021.7	139.3
103.su2cor	Monte-Carlo method	78.4	22794.2	592.7
104.hydro2d	Navier Stokes eqn	96.7	27863.0	619.7
107.mgrid	3D potential field	51.8	18122.4	524.8
110.applu	Partial diff eqn	85.0	10634.1	266.0
125.turb3d	Turbulence modeling	71.3	20249.2	565.4
141.apsi	Weather prediction	98.1	26888.6	592.9
145.fpppp	Quantum chemistry	107.3	31296.7	598.4
146.wave5	Maxwell's eqn	121.1	28792.8	511.4

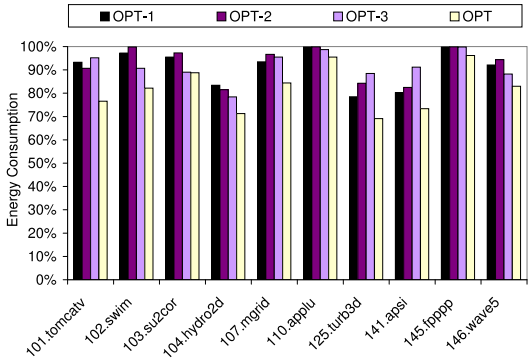


Figure 11: Normalized energy consumptions.

second column of this table gives a brief description of each benchmark. The third column shows the amount of disk-resident data manipulated by each benchmark. The last two columns give the energy consumption and execution cycles for the *BASE* scheme. All the energy and performance results presented in the next section are normalized with respect to the values shown in the last two columns of Table 2.

6.2 Results

We start by presenting an evaluation of *OPT* and its variants. Figure 11 presents energy consumption values for schemes *OPT-1*, *OPT-2*, *OPT-3*, and *OPT*. As mentioned earlier, all the results are normalized with respect to those of the *BASE* scheme. We see from this bar-chart that the average energy improvements achieved by *OPT-1*, *OPT-2*, *OPT-3*, and *OPT* are 8.67%, 7.30%, 8.48%, and 17.95%, respectively. That is, the *OPT* scheme generates significantly better results than the three variants. These results clearly emphasize the importance of tuning all the three components of a disk layout. Starting to stripe every array from the first disk on the disk system prevents the alignment we want to achieve (see the discussion in Section 4). For example, in the *102.swim* benchmark, this causes the energy savings drop from 17.80% to 6.70%. Similarly, fixing stripe size prevents a fine granular clustering, which in turn affects the energy savings in all benchmarks. As an example, looking at the results for the *102.swim* benchmark again, we see that fixing the stripe size causes the energy saving go down from 17.80% to 0.20%. Similar observations can be made for benchmarks *103.su2cor* and *107.mgrid* as well. Finally, when we fix the number of disks over which arrays are striped, we have difficulty in achieving the alignment we want. Consequently, the energy savings drop significantly, as compared to the *OPT* scheme.

The results given in Figure 12 help us compare *OPT*, *HW*, *LOOP*, and *LOOP+OPT*. The first observation one can make from these results is that the *HW* version performs very poorly, mainly because of the short idle periods in disk access traces. The average energy savings brought by this scheme is only 1.67% across the benchmarks in our experimental suite. This can be best explained with the help of the CDF curves given in Figure 13. An (x,y) point on a CDF curve in this graph means that y% of the idle times has a duration of x (ms) or lower, when the layouts in *BASE*/*HW* are used. The minimum amount of idle time required to compensate the cost of spinning down/up the disk is called the *profit threshold*. Based on the numbers from IBM Ultrastar 36Z15, this threshold is 15.19 seconds. The results in this graph show that the idle disk times exhibited by these array/loop-intensive applications are much shorter

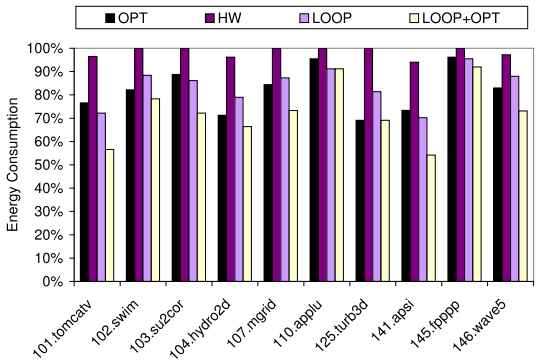


Figure 12: Normalized energy consumptions.

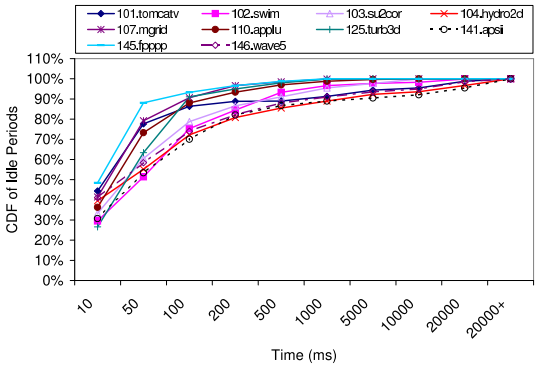


Figure 13: CDF curves for disk idle times, when the layouts in *BASE*/*HW* are used.

than this threshold value, which explains why the *HW* scheme does not achieve much savings. The second observation one can make from Figure 12 is that the *LOOP* scheme is successful in reducing energy consumption. In fact, it achieves an average of 16.07% energy savings, which is very close to the 17.95% savings achieved by *OPT*. Our last observation is that combining loop and data optimizations, that is the *LOOP+OPT* scheme, generates the highest energy savings so far: 27.36% on the average. That is, our disk layout detection approach is very effective when it is combined with code restructuring (transformation). In addition, it needs to be said that the combined scheme evaluated here is actually not a very complex one. One can envision more sophisticated schemes (which is in future agenda) that integrate data and code optimization, which could potentially result in even better energy savings.

Another important metric to consider is the increase in original execution cycles caused by our disk layout based approach. Figure 14 presents the percentage execution time increases brought by *OPT* and *LOOP+OPT* over the *BASE* scheme. We see that the average increase with the *OPT* scheme is only 1.86%, which is not too much at all, considering its large energy benefits. The corresponding increase with the *LOOP+OPT* scheme is 2.61%, when averaged over all the benchmark codes in our experimental suite. The reason that these values are not very high is the fact that our approach tries to strike a balance between performance and energy consumption. As has been discussed earlier in detail, we try to minimize the number of intra-array and inter-array conflicts, and this in turn has a positive impact on performance, and offsets some of the performance overheads incurred by power management. It needs also to be mentioned at this point that an increase in execution cycles can also cause an increase in (leakage) energy consumption of other system components such as caches, main memories, and CPU data-path. However, recall that our application domain is large-scale scientific array codes that manipulate disk-resident data sets. In such applications, energy consumption on the disk system usually dominates the energy consumptions on caches, main memories, and CPU data-path. Therefore, the leakage energy increase due to increased execution cycle count is unlikely to offset the large energy savings coming from the disk system.

In our next set of experiments, we measure energy savings when we change some of the default values used in our experiments so far. We first focus on two parameters: the maximum number of

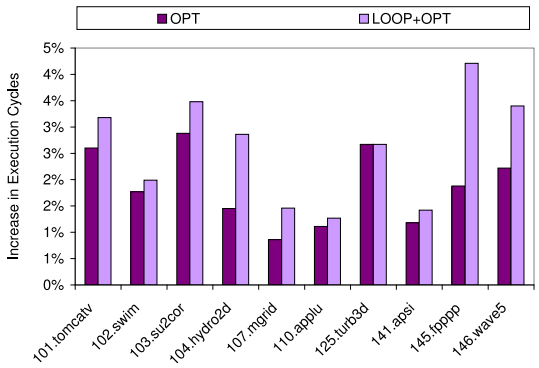


Figure 14: Percentage increase in execution cycles with the *OPT* and *LOOP+OPT*.

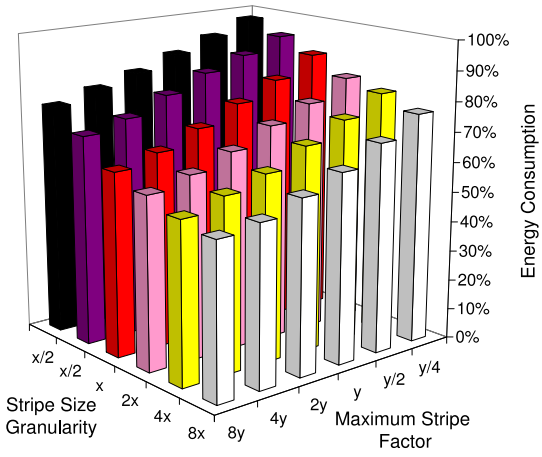


Figure 15: Normalized energy consumption with different values of maximum stripe factor and stripe size granularity.

disks that can be used for striping and the stripe size granularity. An "x" on the x-axis of the graph in Figure 15 corresponds to the default number of stripe sizes that can be selected by our approach (which is 4 as given in Table 1). Also, a "y" on the y-axis denotes the default maximum number of disks that can be used for striping (which is 8 as given in Table 1). The results (the z-axis) represent the average (normalized energy) values over all benchmark codes. We see from these results that our energy savings increase when we increase the number of disks or the stripe size granularity. This can be explained as follows. When the number of disks is increased, it gives more flexibility to our approach for determining the layout and this increases the opportunities for power savings. Similarly, when we have a larger set of stripe sizes to choose from, we can select the most suitable one considering the impact on energy consumption.

We now investigate the influence of the T (threshold) parameter on energy and performance behavior of our profile-driven approach. Recall that this parameter can be used for specifying the percentage of conflicts to eliminate. Therefore, it can be used to study the trade-offs between performance and energy. A higher value (of T) means eliminating more conflicts, which usually results in more disks to store array data, which in turn means less power savings. The default value of T used in our experiments so far was 0.7, as given in Table 1. Figure 16 gives the normalized energy consumption and percentage increase in execution cycles under the different values of the T parameter. All the curves in this figure represent the average values across all ten applications tested. We see from these results that, for both the *OPT* and *LOOP+OPT* schemes, as we reduce the value of T beyond a certain value, the execution cycle overhead curves increase rapidly. Therefore, one may not want to work with very small T values. On the other hand, very large T values may not be very good either, since they increase energy consumption. We see that the values in the middle range such as 0.5, 0.6, and 0.7 tend to strike a good balance between energy savings and performance overheads.

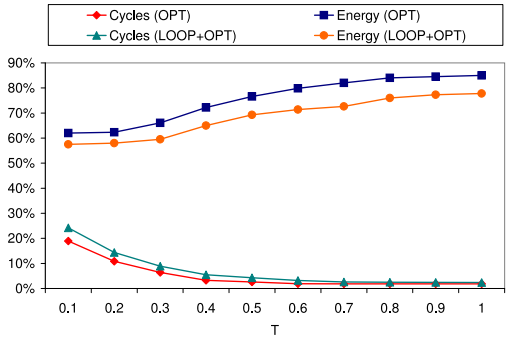


Figure 16: Influence of the T parameter.

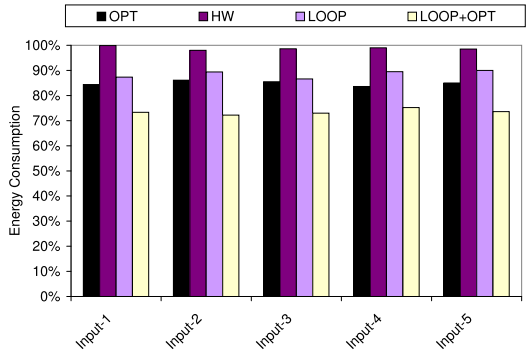


Figure 17: Influence of the input set.

In our last set of experiments, we study the influence of the input set on our results. Since our approach is profile-driven, it is important to study how sensitive our results are to the input set used in profiling. To test this, we performed another set of experiments, where we executed our benchmarks with inputs other than the one used for profiling. We present the results only for the *107.mgrid* benchmark; but, the trends exhibited by the other benchmarks are very similar. Figure 17 presents the normalized energy consumptions for this benchmark with five different inputs (Input-1 through Input-5). However, for all these cases, the disk layouts used are the same and are determined by using the first input set (Input-1). One can see from these results that, while the actual savings can slightly change from one input set to another, the trends are very consistent across the different input sets. This is mainly due to the application domain we target. Recall that we are dealing with array/loop-intensive codes that perform I/O. In these applications, loops are the main control structures, and there are very few conditionally-taken paths. As a result, the input set used does not affect much the flow of execution taken at runtime. Consequently, the disk layouts determined using one set of inputs can be used for executing the application with another set of inputs.

7. CONCLUDING REMARKS AND FUTURE WORK

Disk system is known to be a major contributor to overall power consumption of high-end parallel systems. Past research proposed several architectural level techniques to reduce disk power by taking advantage of idle periods experienced by disks. The main contribution of this paper is a profile-driven approach for determining disk layouts of array data to minimize the energy consumption without increasing overall execution cycles excessively. Specifically, our algorithm determines, for each disk resident array, the stripe factor, stripe size, and start disk for striping. Our experiments with this algorithm reveal that (1) it reduces energy consumption of original applications significantly; (2) the energy savings it generates are competitive with those obtained through loop structuring based strategy; and (3) the best energy savings are achieved by a combined scheme that employs both code and disk layout optimization. Our future work includes designing and implementing algorithms that integrate code and data optimizations for reducing disk power consumption. We are also planning to investigate the impact of such algorithms in other parts of the system such as cache locality and data-path energy. Also in our agenda is extending this work to other application domains.

8. REFERENCES

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

[2] SPEC CPU95 Benchmarks. http://www.spec.org/osg/cpu95/, 1995.

[3] L. Benini, A. Macii, and M. Poncino. Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques. *ACM Transactions on Embedded Computing Systems*, 2(1):5–32, 2003.

[4] L. Benini and G. D. Micheli. System-level power optimization: techniques and tools. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):115–192, 2000.

[5] P. Bohrer, E. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, C. McDowell, and R. Rajamony. *Power-Aware Computing*, chapter The Case for Power Management in Web Servers. Kluwer Academic Publisher, Jan. 2002.

[6] 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.

[7] 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.

[8] 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.

[9] X. Chen and L. Peh. Leakage power modeling and optimization in interconnection networks. In *Proceedings of the International Symposium on Low Power and Electronics Design*, pages 90–95, August 2003.

[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] Y. Fei, L. Zhong, and N. K. Jha. An energy-aware framework for coordinated dynamic software management in mobile computers. In *Proceedings of the IEEE/ACM Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Oct. 2004.

[15] T. D. Givargis, J. Henkel, and F. Vahid. Interface and cache power exploration for core-based embedded system design. In *ICCAD '99: Proceedings of the 1999 IEEE/ACM International Conference on Computer-aided design*, pages 270–273. IEEE Press, 1999.

[16] 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.

[17] 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.

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

[19] 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.

[20] E. J. Kim, K. H. Yum, G. Link, C. R. Das, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Energy Optimization Techniques in Cluster Interconnects. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 459–464. ACM, August 2003.

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

[22] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 6(2):149–206, 2001.

[23] 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.

[24] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *Lecture Notes in Computer Science 768: Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Aug. 1993.

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

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

[27] T. K. Tan, A. Raghunathan, and N. K. Jha. Software architectural transformations: A new approach to low energy embedded software. In *Proceedings of the Design Automation and Test in Europe Conference*, Mar. 2003.

[28] S. U. Wen-Tsong Shiue and C. Chakrabarti. Data memory design and exploration for low-power embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 6(4):553–568, 2001.

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

[30] L. Yan, J. Luo, and N. K. Jha. Combined dynamic voltage scaling and adaptive body biasing for heterogeneous distributed real-time embedded systems. In *Proceedings of the International Conference on Computer-Aided Design*, Nov. 2003.