# An Evaluation of Code and Data Optimizations in the Context of Disk Power Reduction

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

## ABSTRACT

Disk power management is becoming increasingly important in high-end server and cluster type of environments that execute data-intensive applications. While hardware-only approaches (e.g., low-power modes supported by current disks) are successful to a certain extent, one also needs to consider the software side to achieve further energy savings. This paper first demonstrates that conventional data locality oriented code transformations are not sufficient for minimizing disk power consumption. The reason is that these optimizations do not take into account how disk-resident array data are laid out on the disk system, and consequently, fail to increase idle periods of disks, which is the primary metric using which disk power can be reduced. Instead, we propose a disk layout aware application optimization strategy that uses both code restructuring and data layout optimization. Our experimental evaluation with several benchmark codes reveal that the proposed strategy is very successful in reducing disk energy consumption without performing much worse than a pure data locality oriented scheme, as far as execution cycles are concerned. The experiments also show that the benefits coming from our approach increase with the increased number of disks; i.e., it scales very well.

## Categories and Subject Descriptors

D.3.m [**Software**]: Programming Languages—*Miscellaneous*

## General Terms

Languages

## Keywords

disk, code optimization, data optimization, compiler

## 1. INTRODUCTION

Optimizing compiler technology used code and data structuring in the past for reducing execution cycles, reducing memory space requirements, and reducing energy consumption in several system components which include processor data-path, cache memory, and main memory. Recent trends in high-performance computing indicates that disk power is becoming an increasing concern, mainly because many scientific applications executing on such systems are extremely data intensive and frequently access/manipulate disk-resident data sets. While code and data structuring are known to be effective in the past in other parts of the system, their impact on disk system, especially from the perspective of energy consumption, has not been explored by the previous research. The main goal of this paper is to study the impact of code and data optimization on disk system energy. We exclusively focus on scientific applications with regular data access patterns, and try to increase disk inter-access times. The rationale behind is that the increased disk idleness enables more effective exploitation of available power-saving capabilities supported by the underlying disk hardware.

We first focus on loop based code optimization that employs a suite of loop transformations (originally proposed in the context of enhancing data locality and improving loop iteration level parallelism). Our experimental results show that, while pure data locality oriented loop optimizations are effective for some benchmarks, the energy savings achieved by them are not very large. This is mainly because optimizing solely for data locality fails to increase disk idle periods since it does not take into account how disk-resident arrays are laid out on the disk system. Consequently, we make a case for modified code optimizations for increasing energy savings. Our results with the disk-based versions of six randomly-selected, array-based benchmark codes from the Spec2000 suite show that the savings with the modified optimizer are much better than those obtained using a pure data locality optimizer. We then focus on data space and propose a data reorganization scheme (data-to-disk mapping) oriented towards reducing disk energy consumption. An important characteristic of this scheme is that it clusters simultaneously used data in a small number of disks, thereby increasing disk idleness and effective use of available low-power capabilities supported by the architecture. Another characteristic of this layout optimizer is that it is used in conjunction with the code transformation framework presented. Our experimental evaluation with the benchmark codes in our suite indicates that combining code and data transformations under a unified optimizer generates better results than the code optimization alone, as far as disk energy savings are concerned. Overall, our results suggest that, for the best energy savings, any code optimizer should consider the layout of data on the disk system.

The rest of this paper is organized as follows. Section 2 explains the underlying hardware support assumed in this paper for disk power management. Section 3 introduces our experimental platform and presents experimental evidence that show, while a significant fraction of disk energy in scientific applications is spent during idle periods, most of these idle periods are very short to take advantage of. Section 4 discusses our proposed loop transformation framework. Section 5 discusses our data layout optimization. Section 6 presents an experimental evaluation of the proposed code and data optimizations. Section 7 concludes the paper with a summary of its major observations.


*ISLPED'05,* August 8–10, 2005, San Diego, California, USA

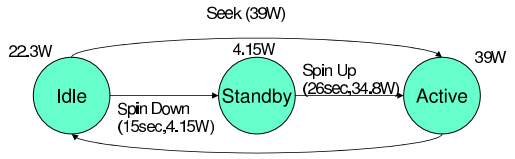


**Figure 1: Disk states and transitions assumed in this work.**

**Table 1: Default simulation parameters.**

| Parameter | Value | Parameter | Value |
|---|---|---|---|
| Processor Speed | 1.5 GHz | Number of Disks | 8 |
| Disk Model | IBM Ultrastar 36Z15 | Interface | SCSI |
| Storage Capacity | 18 GB | RPM | 15,000 |
| Average seek time | 3.4 msec | Average rotation time | 2 msec |
| Internal transfer rate | 55 MB/sec | Power (active) | 13.5 W |
| Power (idle) | 10.2 W | Power (standby) | 2.5 W |

| Parameter | Value |
|---|---|
| Cache Memory | 64KB, 4-way, 128 byte line size |
| 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 |

## 2. DISK POWER MANAGEMENT

Many disks offer several power modes and an idle disk can be transitioned to one of these modes to save power. In a low-power mode, the disk normally stops spinning. Prior studies such as [7, 3, 2] investigated this approach extensively in the context of laptop environments where the primary goal is to save battery power. It needs to be noted that, a disk placed in the low-power mode needs to be spinned up before servicing the next request. Since switching to a low-power mode involves spinning down the disk and servicing the next request involves spinning up the disk, in order for this scheme to be successful, the disk idle periods should be sufficiently large. Otherwise, it does not make sense to switch the disk to the low-power mode. Consequently, detecting idle periods and predicting their durations accurately is critical in this approach. This typically involves some kind of history-based mechanism that records the length and pattern of the idle periods seen up to a specific point during execution.

Figure 1 gives the potential states for a server-class disk that is capable of power management. Each node represents a state (operating mode) and the arrows between the states correspond to state transitions. Attached to each arrow in this figure is the activity that triggers the transition represented by that arrow. The power consumption at each state as well as power/latency values incurred during transitions are also shown in the figure. The values shown in this figure are taken from the data sheet of the IBM Ultrastar 36ZX [6], a server hard disk. In this paper, for all experiments, we assume the model and the values given in Figure 1. Our main goal is to evaluate the potential (disk energy) benefits that could be brought by code and data optimizations. All the different versions of a given application code in our experiments use the same model/values in Figure 1.

## 3. EXPERIMENTAL PLATFORM AND DISK ENERGY CONSUMPTION

To perform our experiments, we wrote a trace generator and developed a disk power simulator. Our trace simulator creates a trace file which is subsequently fed to the simulator. Each entry in the trace includes a time stamp, type of activity (read/write), amount of data read/written, and duration of activity. The cycle estimates for the loop nests were obtained from the actual execution of the programs on a SUN Blade1000 machine (UltraSPARC-III architecture operating at 750 MHz with Solaris 2.9) and these estimates are used in all our simulations. In addition to the I/O trace file, the disk simulator needs the disk layout information for each array, which includes the number of disks and stripe size. Using disk layout parameters and traces, 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.

**Table 2: Six SPEC 2000 benchmarks used in this study.**

| Benchmark Name | Brief Description | Data Size (MB) | Energy Cnsmp (J) | Execution Time (ms) |
|---|---|---|---|---|
| 177.mesa | 3D Graphics Library | 181.4 | 16322.7 | 21018.0 |
| 178.galgel | Computational Fluid Dynamics | 216.6 | 24055.5 | 29101.2 |
| 183.equake | Seismic Wave Propagation | 337.4 | 30486.7 | 36049.5 |
| 188.ammp | Computational Chemistry | 195.5 | 20872.4 | 27006.2 |
| 191.fma3d | Finite Element Crash Simulation | 298.1 | 27306.2 | 33156.9 |
| 301.apsi | Meteorology: Pollutant Distribution | 354.1 | 42388.2 | 47751.1 |

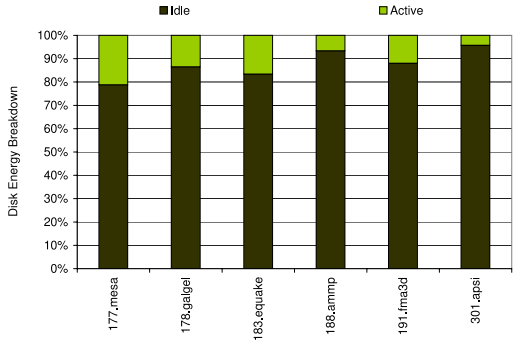


**Figure 2: Disk energy breakdown for the original codes.**

The code modifications necessary for the schemes tested in this paper are implemented using the SUIF compiler [5]. While these modifications increase the compilation times by about 50% on the average (across all benchmark codes used), our belief is that this increase is within tolerable limits, in particular when one considers the large energy savings achieved.

Table 2 gives the benchmark codes used in this study. These benchmark codes (all of which are array based) are selected randomly from the Spec 2000 floating-point benchmark suite. In order to test our approach, we made the data arrays used by these benchmarks *disk resident.* Specifically, each array is stored in a file, which in turn is stored on the disk system. Since simulating disk activity is extremely slow, in our simulations we skipped the first 250 million instructions and simulated the next 1 billion instructions from each benchmark. The second column of Table 2 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 disk energy consumption and execution cycles for the original benchmarks with disk-resident data sets when *no* disk power management is employed. All the energy/performance improvements presented in the rest of this paper are *normalized* with respect to the values shown in the last two columns of Table 2.

Figure 2 gives the breakdown of disk energy consumption between idle periods and active periods when *no* power management strategy is employed. One can observe from this bar-chart that a significant fraction of total disk power, about 87.62% on the average to be specific, is spent in idle periods. While, looking at these results in Figure 2, one might think that the disks in the system have long idle periods, our experiments also showed that the disk idle periods are not large. In fact, most of the idle periods are very small. Therefore, we can conclude that the idle periods constitute a large portion of overall disk power due to numerous very small idle periods, rather than a few large idle periods.

Unfortunately, these results are not very good from an energy management angle since conventional (hardware based) disk power management techniques work best with long idle periods. There are two ways of getting around this problem. Firstly, one can develop new power optimization schemes that can take advantage of these small idle periods. Studies such as [1] and [4] pursue this option. The second option is to modify default code structure and/or disk layout of arrays so that idle periods become longer. This is the option explored in this paper.

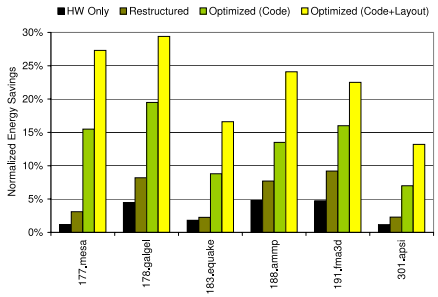


**Figure 3: Percentage energy savings with different schemes.**

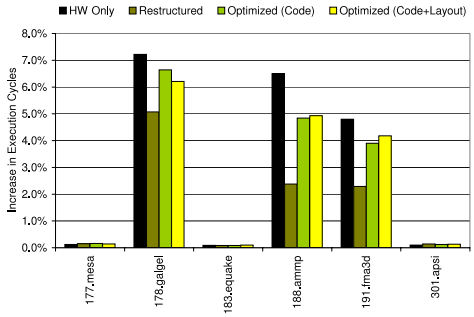


**Figure 4: Percentage increase in original execution cycles with different schemes.**

At this point, one may think that conventional data locality oriented code optimizations can help increase idle periods for disks. To check whether this is indeed the case, we performed another set of experiments, this time each benchmark is optimized using a data locality optimizer whose main target is to maximize data locality. The specific optimizer used for obtaining the restructured codes is based on the idea of exploiting as much data reuse as possible within the innermost loop positions. Note that this is beneficial from both cache and page level data locality perspectives. The optimizer first captures data reuse information and represents it in form of a data reuse matrix. It then applies a series of transformations to bring the data reuse matrix into an optimized form, which means ensuring that data reuses occur in the innermost loop iterations. After this step, the approach uses loop fusion [10] and loop tiling [10] to further improve temporal data reuse. The details of this code restructuring scheme are not very important as far as the focus of this paper is concerned. But, we need to mention that it represents state-of-the-art in data locality optimization.

When we compare the energy savings achieved by the hardware-only approach (i.e., an approach that uses past history to spin down a disk when predicted idleness is sufficiently large) and this locality-aware restructuring based scheme (see the first two bars for each benchmark in Figure 3), we do not see much improvements. Specifically, the average energy savings achieved by the two versions are 3.03% and 5.46%. The bar-chart in Figure 4 gives the percentage increases in the original execution cycles. We see that the results with the restructuring based scheme (the second bar for each benchmark) are slightly better than the results with the hardware-only scheme (the first bar for each benchmark) as a result of optimized data locality. Overall, these results indicate that restructuring an application code just based on data locality (i.e., without paying attention to how the array data is laid out on the disk system) is not very effective from the energy saving angle.

# 4. CODE OPTIMIZATION FOR DISK POWER REDUCTION

In this section, we present our code restructuring approach to disk power optimization. We focus on three optimizations (loop fusion/fission, loop tiling, and linear optimizations), and also discuss how they can be combined under a unified optimizer that targets at disk power management.

## 4.1 Loop Fusion and Fission

Combining two loops into a single loop is called loop fusion. It is typically used to bring array references to the same elements close together [10]. Consider the code fragment on the left side of the example below written using a C-like notation, which consists of two separate loops (nested within an outer loop) that access the same array ($X_1$).

$$\begin{array}{l} \text{for}(j=0; j<L_j; j++) \\ \{ \\ \quad \text{for}(i=0; i<L_i; i++) \\ \quad\quad X_1[i] = i * i + s_1; \\ \quad \text{for}(i=0; i<L_i; i++) \\ \quad\quad s_2 = s_2 + X_1[i] * X_1[i]; \\ \} \end{array} \quad \Rightarrow \quad \begin{array}{l} \text{for}(j=0; j<L_j; j++) \\ \quad \text{for}(i=0; i<L_i; i++) \\ \quad \{ \\ \quad\quad X_1[i] = i * i + s_1; \\ \quad\quad s_2 = s_2 + X_1[i] * X_2[i]; \\ \quad \} \end{array}$$

If the loop limit is sufficiently large that array $X_1$ does not fit in cache, this code will stream the array from memory through the cache twice (once for each loop). If this fragment is transformed into the form shown on the right hand side, the array needs to be streamed through the cache only once since its contribution to the second assignment can be calculated, while the cache line holding $X_1[i]$ is still cache resident from its use in the first assignment statement. This simple example illustrates that loop fusion can improve cache-level data locality by bringing accesses to the same array closer. A similar argument can be made for page-level data locality as well.

From the viewpoint of disk power however, one needs to be careful in applying this transformation. In the above example, depending on the loop bound $L_i$, the idleness experienced by each disk would be small. Therefore, fusing the two $i$ loops can increase disk idleness and thus beneficial from the disk energy perspective as well. However, if the loop nests to be fused contain extra arrays (i.e., arrays that are not targeted by fusion), these arrays might lead to accesses to a large number of disks (some of which would not be accessed had we not fused the loops). Therefore, in a parallel disk architecture, loop fusion should be applied with care. One criterion in applying this optimization is to check whether fusing loops would lead to activation of more disks than individual nests demand. Loop fission (also known as loop distribution [10]) is the reverse of loop fusion, and 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 the 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.

It is important to note the conflicting objectives of minimizing data locality and optimizing disk energy consumption when these transformations are employed. In general, when one wants to optimize data data locality, loop fusion is preferable whereas loop distribution is generally used to enhance iteration-level parallelism by placing the sinks and sources of data dependences into separate loops. As far as disk energy minimization is concerned, however, loop fission is, in general, preferable as it has the capability of isolating accesses to small set of disks. As an example, let us consider the following transformation:

$$\begin{array}{l} \text{for}(i=0; i<L_i; i++) \\ \{ \\ \quad s_1 = X_1[i] + s_2 * X_2[i]; \\ \quad s_3 = s_3 + X_3[i] * X_4[i]; \\ \} \end{array} \quad \Rightarrow \quad \begin{array}{l} \text{for}(i=0; i<L_i; i++) \\ \quad s_1 = X_1[i] + s_2 * X_2[i]; \\ \text{for}(i=0; i<L_i; i++) \\ \quad s_3 = s_3 + X_3[i] * X_4[i]; \end{array}$$

This transformation separates accesses to arrays $X_1$ and $X_2$ from accesses to arrays $X_3$ and $X_4$. Consequently, if the first two arrays are stored in a different set of disks than the last two arrays, this transformation can increase disk idleness, thereby leading to a more effective low-power management of the disk system.

Based on the discussion above, we propose the following strategy for applying loop fusion and fission in a parallel disk environment. If there is no cache/main memory related data locality

concern, then we do not apply loop fusion; we apply loop fission in such a way that the arrays that share the same set of disks reside within the same loop after fission. If there is a data locality concern, we do not modify our loop fissioning strategy except that we do not separate statements that contain references to the same array (in an attempt to preserve data locality). Our fusioning/fissioning strategy tries to strike a balance between two objectives (optimizing for disk power versus optimizing for data locality in cache/memory). When applying loop fusion in a cache-based environment, we take cache considerations into account but never fuse two loops if doing so increases the number of disks accessed in a single iteration. For example, suppose that there are three one-dimensional fussable loops in the code, each with one statement within it: $s_1 += X_1[i]+X_2[i]$ in the first loop; $s_2 += X_1[i+1] * X_2[i-1]$ in the second loop; and $s_3 += X_3[i] - X_2[i]$ in the third loop ($i$ is the loop index). Also, assume that each array is stored in a separate disk. In this case, while a pure cache locality-oriented approach would fuse all three loops (in conjunction with array padding), our disk power conscious approach would fuse only the first two loops. As in the case of loop fission, this loop fusion scheme also tries to find a balance between conflicting objectives. To sum up, in a data locality sensitive environment, we use cache/memory constraints to restrict loop fission and disk constraints (e.g., minimizing the number of active disks) to restrict loop fusion.

## 4.2 Loop Tiling

A widely-used technique for improving data locality is loop tiling [10]. In tiling, data structures that are too big to fit in the cache (or the highest level of memory under consideration) are broken up into smaller pieces that will fit in the cache. Consider the following matrix-multiply example (left hand side below). If the arrays accessed in this nest do not fit in the cache, the cache performance might be poor.

```
for(i = 0; i < L_i; i++)                    for(ii = 0; ii < L_i; ii = ii + T)
  for(j = 0; j < L_j; j++)                    for(jj = 0; jj < L_j; jj = jj + T)
    for(k = 0; k < L_k; k++)          ⇒         for(i = ii; i < ii + T; i++)
      X3[i][j] += X1[i][k] * X2[k][j];            for(j = jj; j < jj + T; j++)
                                                    for(k = 0; k < L_k; k++)
                                                      X3[i][j] += X1[i][k] * X2[k][j];
```

If this nest is tiled (blocked) as shown on the right hand side (assuming that the tile size, $T$, divides loop bounds $L_i$ and $L_j$ evenly), a square-block of array $X_3$ is computed by taking the product of a row-block of $X_1$ with a column-block of $X_2$. This product consists of a series of sub-matrix multiplies. If these three blocks, one from each matrix, all fit in cache simultaneously, their elements only need to be read in from memory once for each sub-matrix multiply. Thus, the array $X_1$ will now only need to be touched once for each column-block of $X_3$, and $X_2$ will only need to be touched once for each row-block of $X_1$. As a result, the memory traffic will be reduced by the size of the blocks.

While this transformation enhances temporal locality across multiple loop levels, it also modifies the array access pattern dramatically. For instance, after the transformation, at a given time, a column-block of array $X_2$ is active. We observe that depending on the tile size parameter, a majority of these elements are not consecutive in data space (assuming a row-major array layout). Consequently, all the disks that hold these elements need to be active during a given short period of time.

Our disk power-aware tiling strategy works as follows. It first determines the loops that carry some form of data reuse since applying tiling to a loop which does not carry any reuse does not promote data reuse but only increases loop overhead. We achieve this using the reuse-oriented tiling strategy proposed by Xue and Huang [11]. Then, among these loops (with data reuse), it selects a subset such that the resulting access pattern does not generate a data tile (i.e., data footprint) on the array space which is orthogonal to the storage direction of the array. This is because, under the assumption that elements of a given array are stored consecutively in the data space (from the first element to the last element), a data tile orthogonal to the storage direction (of the array) leads to a maximum number of disk activation. For example, in a two-dimensional row-major array case, the disk power aware tiling strategy never selects an iteration space tile shape if it leads to a column-block data tile on the array space. If possible, it works with only row-block and square tiles. In the ideal case, one would want to work with only row-block data tiles (for such an array); but, in many cases, due to data dependences and array access patterns, it may not be possible to obtain only row-block tiles. Still, our experiments show that many nested loops can be tiled using only row-block and square tiles. To achieve this, when necessary, linear loop optimizations such as loop permutation can be used prior to tiling. To sum up, our tiling strategy first determines the loops with data reuse, then filters out the ones with orthogonal footprints (with respect to the storage order), and after that, tiles the resulting nest. Our current implementation also tries all permutations of outer nests to obtain row-block and square tiles (i.e., eliminate column-block tiles).

| Array Allocation on Disks | | | | Poor Data Locality: $[j][i]$ | | Good Data Locality: $[i][j]$ | |
|---|---|---|---|---|---|---|---|
| $d_0$ | $d_1$ | $d_2$ | $d_3$ | Cacheless | Cache | Cacheless | Cache |
| $X_1, X_2, X_3, X_4$ | × | × | × | × | linear | × | × |
| $X_1, X_2$ | $X_3, X_4$ | × | × | fission | fission + linear | fission | × |
| $X_1, X_2, X_3$ | $X_4$ | × | × | × | linear | × | × |
| $X_1$ | $X_2$ | $X_3$ | $X_4$ | fission | fission + linear | fission | × |

**Figure 5: Selection of different loop optimizations based on access pattern and array allocation. Symbol × in the first four columns indicate that no array is mapped to the corresponding disk. Symbol × in the last four columns indicate that no optimization. $d_0$, $d_1$, $d_2$, and $d_3$ are the disks.**

## 4.3 Linear Loop Transformation

Linear loop transformations that aim at improving data locality generally try to achieve either of two objectives for each array reference: optimizing temporal locality in the innermost loop or optimizing spatial locality in the innermost loop. Optimizing temporal locality in the innermost loops allows the back-end compiler to place the reference in question into a register (provided that no aliasing occurs). This eliminates accesses to the data space, thereby increasing the disk idle times and creating more opportunities for the employment of low-power operating modes. Optimizing spatial locality (unit stride accesses) is beneficial from the disk power reduction angle as well since it allows all the accesses to a given disk to be completed before moving to another disk (provided that the array elements are stored sequentially).

It needs to be emphasized however that there are cases where linear transformations might be desirable from one objective's angle but not desirable from the other's angle. Consider the following nested loop which accesses a two-dimensional row-major array:

```
for(i = 0; i < L_i; i++)
  for(j = 0; j < L_j; j++)
    X1[j][i] = X1[j][i] * X1[j][i] - 1;
```

Since the column-wise access pattern exhibited by the inner loop here is not suitable from the data locality perspective, a solution is to interchange the order of the loops. Such an optimization makes the accesses in the inner loop consecutive in memory, and consequently improves data locality. Assuming now that array $X_1$ spans multiple disks, the loop interchange here is beneficial from the disk energy perspective as well. This is because, after the interchange, the array is accessed sequentially; that is, all array accesses to a disk are completed before moving to the next disk. However, if we assume that the entire array fits into a single disk, then an energy-oriented optimization strategy would not need to perform any transformation since no transformation would have an effect on the inter-access time of the disk in question. However, from a data locality point of view, it is still desirable to apply loop interchange.

Our disk power-conscious linear loop transformation strategy works as follows. If there is no data locality concern, the compiler

```
Disk-Conscious-Fusion(N)
INPUT: N = N_1, N_2, ..., N_s,
   candidate nests for fusion
ALGORITHM:
build M = {M_1, ..., M_t} where:
 M_i = {m_i} is compatible nest set,
 and depth(M_{i+1}) ≤ depth(M_i);
build DAG H with dependence edges and weights;
for each M_i = {m_1, ..., m_p} {
 for k_1 = m_1 to m_p {
  for k_2 = m_2 to k_1 {
   if (no cache memory) continue;
   if ((there exists locality between k_1 and k_2)
   and (Disks(Arrays(k_1)) == Disks(Arrays(k_2)))
   and (it is legal to fuse k_1 and k_2))
    fuse k_1 and k_2 and update H;
  }
 }
}
```

**Figure 6: Disk-conscious loop fusion algorithm.**

```
Disk-Conscious-Fission(N)
INPUT: N = N_1, N_2, ..., N_s,
   candicate nests for fission
ALGORITHM:
for each N_i = {n_1, ..., n_k},
    where n_j s are individual loops in N_i {
 let p_1, ..., p_l be the statements in N_i;
 for each n_j ∈ N_i, j = 1, k {
  if (no cache memory) {
   distribute n_j over n_{j+1}, ..., n_k, p_1, ..., p_l
    such that:
    if (Disks(Arrays(p_k)) == Disks(Arrays(p_j))) then
     p_k and p_j stay in the same loop after distribution;
  } else {
   apply classical (performance-oriented) loop distribution
    algorithm such that:
    if (Disks(Arrays(p_k)) == Disks(Arrays(p_j))) then
     p_k and p_j stay in the same loop after distribution;
  }
 }
}
```

**Figure 7: Disk-conscious loop fission (loop distribution) algorithm.**

```
Disk-Conscious-Optimization(N)
INPUT: N = N_1, N_2, ..., N_s,
   nests in the procedure
ALGORITHM:
Disk-Conscious-Fission(N);
Disk-Conscious-Fusion(N);
for each N_i = {n_1, ..., n_k},
    where n_j s are individual loops in N_i {
 best-cost = ∞;
 best-permutation = none;
 determine permutations of n_1, ..., n_k
  with the best locality;
 let P_1, ..., P_f be such permutations;
  for each P_i, i = 1, f {
   current-cost = find the number of disks accessed
    by the arrays with no locality;
   if (current-cost < best-cost) {
    best-cost = current-cost;
    best-permutation = P_i;
   }
  }
 determine the set S_i, the loops with reuse in P_i;
 if (there is a cache in the system)
  tile each loop s_j ∈ S_i if its data footprint is not
   orthogonal to storage direction;
}
```

**Figure 8: Disk-conscious energy optimization algorithm.**

tries to optimize spatial and temporal locality aggressively. Specifically, it uses the loop transformation framework presented in [8]. However, it does not apply a transformation if the transformation will not reduce the number of active disks at a time or cluster array accesses (e.g., when the array fits in a single disk). If data locality is a concern, it tries to optimize locality taking cache/main memory characteristics into account, and take disk power into account only when it needs to distinguish between references with no data locality. As an example, suppose that a nested loop that manipulates three arrays ($X_1$, $X_2$, and $X_3$) can be optimized for locality in two alternate ways (using linear loop transformations). In the first alternative, arrays $X_1$ and $X_2$ have unit stride accesses, whereas array $X_3$ has no data locality. In the second alternative, arrays $X_1$ and $X_3$ have unit stride accesses but array $X_2$ has no data locality. Then, our strategy calculates how many different disks are accessed due to array $X_3$ in the first alternative and due to array $X_2$ in the second alternative. It selects the alternative that accesses the minimum number of disks.

## 4.4 Putting It All Together

So far we have considered our code optimizations in isolation. When we consider the interaction between these optimizations, the problem becomes much harder. In particular, it should be noted that the two objective functions, namely, improving data locality and reducing disk energy consumption can demand different combinations of transformations. To demonstrate this point, we consider the following nested loop which accesses four different disk-resident arrays:

```
for(i = 1; i < L; i++)
  for(j = 1; j < L; j++)
    {
      X_1[i][j] = X_2[i][j] + 1;
      X_3[i][j] = X_4[i][j] - 1;
    }
```

Let us assume that arrays $X_1$ and $X_2$ are stored in one disk, and arrays $X_3$ and $X_4$ reside in another disk. A pure data locality oriented optimization scheme would normally not perform any transformation on this loop, since all the references exhibit high spatial locality and the loop body is not large enough to justify loop distribution (due to instruction cache locality concerns). A disk power oriented strategy, on the other hand, would apply loop distribution to isolate the accesses to individual disks so as to maximize the idle periods for each disk. Now, let us assume that all the subscript expressions in the last example above are $[j][i]$ instead of $[i][j]$ (under the same array placement scheme). In this case, a locality-oriented optimization strategy would apply loop interchange to obtain unit stride accesses in the inner loop position. A strategy that targets at disk energy would, however, still use loop distribution. If optimizing both disk energy and data locality is important in this example, then it would be best to apply both loop interchange and loop distribution. In this last scenario, if all the arrays reside on the same disk, data locality optimization would demand a loop interchange, whereas disk optimization would require no transformation. This example demonstrates that the selection of loop transformations to apply depends strongly on the data locality characteristics of the code as well as the array allocation in the disk system (i.e., array-to-disk mapping). Figure 5 gives the optimizations to be applied (considering only loop fission and linear loop transformation) for the example above, assuming a disk system with 4 disks. The first four columns in this figure give the array allocation on the disk system. The fifth and sixth columns cover the case when array subscript expressions (for all arrays) are of the form $[j][i]$; i.e., they exhibit poor data locality, whereas the seventh and eight columns correspond to the case with subscript expressions $[i][j]$ (that is, good data locality). For each case (good or poor locality), we consider a cacheless system and a system with a data cache. From this table, we clearly see that both array allocation (placement) on the disk system and locality play a role in determining the optimization(s) to be applied.

An important issue then is to combine our loop-based transformations in such a fashion that both the disk energy and the data locality are optimized. Our heuristic strategy to this problem operates as follows. We first apply loop fission to isolate as many nested loops as possible. This will enable the compiler to spin down as many disks as possible. After that, we apply disk power conscious version of loop fusion to take advantage of cache memory (if there is one in the system). Then, we consider each of the resulting loop nests one-by-one, and optimize them using disk power conscious versions of loop permutation (linear transformation) and loop tiling. Figure 8 shows the overall compiler algorithm. This algorithm calls the algorithms Disk-Conscious-Fusion(.) and Disk-Conscious-Fission(.), given in Figures 6 and 7, respectively. The algorithm in Figure 6 is a greedy heuristic based on the depth of compatibility, similar to the performance-oriented loop fusioning strategy presented in [9]. It builds a DAG from candidate loops, where edges are dependences between the loops and the weight of each edge is the potential gain due to loop fusion. The nests are partitioned into sets of compatibility at the deepest loop levels possible. Note that the approach first fuses the nests with the deepest compatibility and locality. Then, the DAG is updated and the fusion is applied at the next level until all compatible sets are

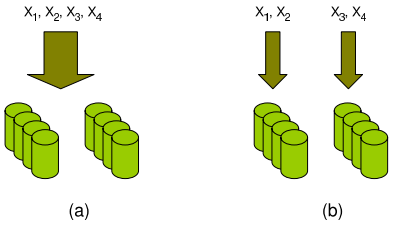


**Figure 9: Two different data allocations on the disk system.**

considered. The algorithm in Figure 7, on the other hand, considers each loop nest one-by-one, and applies loop distribution while being careful in not distorting data locality. In both the algorithms, for a given loop $l$, Arrays($l$) gives the set of arrays accessed by it and Disks(Arrays($l$)) gives the set of disks touched. After applying loop fission and fusion, within the outer for-loop (in Figure 8), each of the nests is optimized using loop permutation and tiling for disk energy and data locality.

# 5. DATA OPTIMIZATION FOR DISK POWER REDUCTION

While the disk layout-aware code structuring strategy presented above increases disk idle times, one can do even better by selecting the most appropriate disk layout for each array (following the code structuring). What we mean by layout optimization in this section is to determine the most appropriate set of disks to store a given disk-resident array. Let us assume a code fragment with two loop nests, one of the accessing arrays $X_1$ and $X_2$, whereas the second one accessing arrays $X_3$ and $X_4$. If all the four arrays are striped over all the disks in the system (see Figure 9(a)), this means all the disks will be active in both the nests, thereby presenting very little opportunity for power management. In comparison, if $X_1$ and $X_2$ are stored in half of the disks and $X_3$ and $X_4$ are stored in the remaining disks (see Figure 9(b)), only half of the disks will be active at any given nest, meaning that the remaining disks can be spun down. Our data layout optimization scheme is applied right after code structuring (discussed earlier), and determines a suitable set of disks for each array based on the principle shown in Figure 9. We do not give the pseudo code of the formal algorithm due to space concerns.

# 6. EXPERIMENTAL EVALUATION

We now present an experimental analysis of the code and data layout optimization approach discussed in earlier sections. Let us first look at the energy savings achieved by the two versions of our approach. In the first one, we apply only code restructuring, and in the second one, we apply both code and data restructuring. The results are given in Figure 3 as the third and fourth bar for each benchmark code. We see from these results that the energy savings achieved by these two versions are 13.38% and 22.18%, that is, they are much better than those obtained by the hardware-only approach and the pure data locality oriented code restructuring (the first two bars in the same figure). Also, since the gap between the two versions of our approach is not small, we can conclude that considering data allocation (layout) on the disk system is important for achieving high energy savings. The performance results of these two versions (given as the last two bars for each benchmark in Figure 4) reveal that, while they are not as good as the pure data locality oriented scheme, they are not too far from it either.

One of the important parameters whose impact on the disk power needs to be studied is the number of disks in the system. Recall that our default configuration has 8 disks. Figure 10 gives the energy saving results with varying number of disks (x-axis), when averaged over all six codes in our suite. Maybe the most important trend that can be observed from these curves is that our approach is able to take advantage of additional disks in the system (as far as energy saving is concerned). In fact, we see that the gap between our approach and the other two increases as we increase the number of disks in the system. Considering that, in parallel to the increase in dataset sizes and in data duplication due to reliability reasons, the number of disks employed normally keeps increasing, these results are promising.

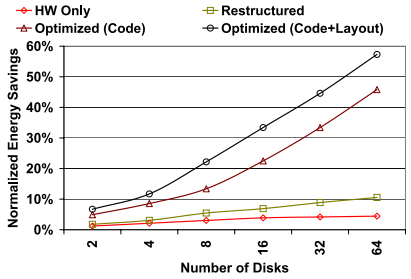


**Figure 10: Impact of the number of disks.**

# 7. CONCLUSIONS

As power consumption of disk systems of large-scale servers/clusters is becoming an increasing concern (due to thermal issues and cooling costs), we need both hardware and software solutions for addressing the problem. This paper proposes and evaluates a compiler-guided optimization framework for reducing disk energy consumption. An important characteristic of the proposed framework is that it combines both code and data optimizations under a unified optimizer. The experimental results with six applications show that this approach saves much more energy than a pure hardware-oriented scheme that does not employ any software optimization and a pure data locality optimizer that does not take into account the layout of array data on the disk system. The experimental results also indicate that the proposed optimization framework scales well as we increase the number of disks in the system.

# 8. REFERENCES

[1] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving disk energy in network servers. In *Proc. of the 17th International Conference on Supercomputing*, pages 86–97. ACM, June 2003.

[2] F. Douglis, P. Krishnan, and B. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proc. of the 2nd Symposium on Mobile and Location-Independent Computing*, pages 121–137, 1995.

[3] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the power-hungry disk. In *Proc. of the USENIX Winter Conference*, pages 292–306, 1994.

[4] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. Drpm: Dynamic speed control for power management in server class disks. In *Proc. of the International Symposium on Computer Architecture*, pages 169–179, June 2003.

[5] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, Dec. 1996.

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

[7] R. K. K. Li, P. Horton, and T. Anderson. A quantitative analysis of disk drive power management in portable computers. In *Proc. of the USENIX Winter Conference*, pages 279–292, 1994.

[8] W. Li. Compiling for numa parallel machines. In *Ph.D. Thesis*. Department of Computer Science, Cornell University, 1993.

[9] K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 1996.

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

[11] J. Xue and C.-H. Huang. Reuse-driven tiling for data locality. In *Z. Li et al. [Eds.], Lecture Notes in Computer Science*, page Volume 1366. Springer-Verlag, 1998.