Improving I/O Performance of Applications through Compiler-Directed Code Restructuring

Mahmut Kandemir Seung Woo Son
Department of Computer Science and Engineering
The Pennsylvania State University
$\{$*kandemir,sson*$\}$*@cse.psu.edu*

Mustafa Karakoy
Department of Computing
Imperial College
m.karakoy@imperial.edu.uk

Abstract

Ever-increasing complexity of large-scale applications and continuous increases in sizes of the data they process make the problem of maximizing performance of such applications a very challenging task. In particular, many challenging applications from the domains of astrophysics, medicine, biology, computational chemistry, and materials science are extremely data intensive. Such applications typically use a disk system to store and later retrieve their large data sets, and consequently, their disk performance is a critical concern. Unfortunately, while disk density has significantly improved over the last couple of decades, disk access latencies have not. As a result, I/O is increasingly becoming a bottleneck for data-intensive applications, and has to be addressed at the software level if we want to extract the maximum performance from modern computer architectures.

This paper presents a compiler-directed code restructuring scheme for improving the I/O performance of data-intensive scientific applications. The proposed approach improves I/O performance by reducing the number of disk accesses through a new concept called disk reuse maximization. In this context, disk reuse refers to reusing the data in a given set of disks as much as possible before moving to other disks. Our compiler-based approach restructures application code, with the help of a polyhedral tool, such that disk reuse is maximized to the extent allowed by intrinsic data dependencies in the application code. The proposed optimization can be applied to each loop nest individually or to the entire application code. The experiments show that the average I/O improvements brought by the loop nest based version of our approach are 9.0% and 2.7%, over the original application codes and the codes optimized using conventional schemes, respectively. Further, the average improvements obtained when our approach is applied to the entire application code are 15.0% and 13.5%, over the original application codes and the codes optimized using conventional schemes, respectively. This paper also discusses how careful file layout selection helps to improve our performance gains, and how our proposed approach can be extended to work with parallel applications.

1 Introduction

In the recent past, large scale applications in science and engineering have grown dramatically in complexity. As a result, scientists and engineers expend great effort to implement software systems that carry out these applications and interface them with the instruments and sensors that generate data. Apart from their huge computational needs, these large applications have tremendous I/O requirements as well. In fact, many scientific simulations tend to generate huge amounts of data that must be stored, mined, analyzed, and evaluated. For example, in a combustion application [39], features based on flame characteristics must be detected and tracked over time. Based upon evolution, simulations need to be steered in different regions, and different types of data need to be stored for further analysis. A simulation involving three-dimensional turbulent flames involving detailed chemistry can easily result in 5 tera-bytes of data being stored on disk, and the total storage requirement can be in the order of peta-bytes when one considers the fact that numerous such simulations have to be performed to reach meaningful and accurate conclusions. Other scientific applications have also similar storage and I/O requirements.

Unfortunately, as far as software – in particular compilers – are concerned, I/O has always been neglected and received much less attention in the past compared to other contributors to an application's execution time, like CPU computation, memory accesses and inter-CPU communication. This presents an important problem, not just because modern large-scale applications have huge I/O needs, but also the progresses in storage hardware are not in the scale that can meet these pressing I/O demands.

Advances in disk technology have enabled the migration of disk units to 3.5-inch and smaller diameters. In addition, the storage density of disks has grown at an impressive 60 percent annually, historically, and has accelerated to greater than a 100 percent rate since 1999 [20]. Unfortunately, disk performance has not kept pace with the growth in disk capacities. As a result, I/O accesses are among primary bottlenecks in many large applications that store and manipulate large data sets. Overall, huge increases in data set sizes combined with slow improvements in disk access latencies motivate for software-level solutions to the I/O problem. Clearly, this I/O problem is most pressing in the context of data-intensive scientific applications, where increasingly larger data sets are processed.

While there are several ways of improving I/O behavior of a large application, one of the promising approaches has been cutting the number of times the disks are accessed during execution. This can be achieved at different layers of the I/O subsystem and be attacked by using caching which keeps frequently used data in memory (instead of disks) or by restructuring the application code in a way that maximizes data reuse. While both the approaches have been explored in the past [2, 3, 9, 10, 16, 21, 24], the severity of the I/O problem discussed above demands further research. In this paper, we focus on a compiler-directed code restructuring for improving I/O performance of large-scale scientific applications that process disk-resident data sets. A unique advantage of the compiler is that it can analyze an entire application code, understand global (application wide) data and disk access patterns (if data-to-disk mapping is made available to it), and – based on this understanding – restructure the application code and/or data layout to achieve the desired performance goal. This is a distinct advantage over pure operating system (OS) based approaches that employ rigid, application agnostic optimization policies as well as over pure hardware based techniques that do not have the global (application wide) data access pattern information. However, our compiler based approach can also be used along with OS and hardware based schemes, and in fact, we believe that this is necessary to reach a holistic solution to the growing I/O problem.

The work presented in this paper is different from prior studies that explore compiler support for I/O in at least two aspects. First, our approach can optimize the entire program code rather than individual, parallel loop-nests, as has been the case with the prior efforts. That is, as against to most of the prior work on compiler-directed I/O optimization, which restructure loops independent of each other, our approach can restructure the entire application code by capturing the interactions among different loop nests. An advantage of this is that our approach does not perform a local (e.g., loop nest based) optimization which is effective for the targeted scope but harmful globally. However, if desired, our approach can be applied to individual loop nests or functions/subprograms independently. Second, we also discuss the importance of file layout optimization and of adapting to parallel execution. These two extensions are important as 1) the results with our layout optimization indicate that additional performance savings (7.0% on average) are possible over the case code re-structuring is used alone, and 2) the results with the multi-CPU extension show that this extension brings 33.3% improvement on average over the single-CPU version.

The proposed approach improves I/O performance by reducing the number of disk accesses through *disk reuse maximization.* In this context, disk reuse refers to reusing the data in a given set of disks as much as possible before moving to other disks. Our approach restructures the application code, with the help of a polyhedral tool [26], such that disk reuse is maximized to the extent allowed by intrinsic data dependencies in the code. We can summarize the major contributions of this paper as follows:

• We present a compiler based disk reuse optimization technique targeting data intensive scientific applications. The proposed approach can be applied at the loop nest level or whole application level.

• We discuss how the success of our approach can be increased by modifying the storage layout of data, and how it can be extended to work under parallel execution.

• We present an experimental evaluation of the proposed approach using seven large scientific applications. The results collected so far indicate that our approach is very successful in maximizing disk reuse, and this in turn results in large savings in I/O latencies. More specifically, the average I/O improvements brought by the loop nest based version of our approach are 9.0% and 2.7%, over the original application codes and the codes optimized using conventional schemes, respectively. Further, the average improvements obtained when our approach is applied to the entire application code are 15.0% and 13.5%, over the original application codes and the codes optimized using conventional schemes, respectively.

The rest of this paper is organized as follows. The next section explains the disk system architecture assumed by our compiler. It also presents the key concepts used in the remainder of the paper. Section 3 gives the mathematical details behind the proposed compiler-based approach. Section 4 discusses how our approach can be extended by taking accounts of the storage layout of data. Section 5 gives an extension to capture the disk access interactions among the threads of a parallel application. An experimental evaluation of our approach and a comparison with the conventional data reuse optimization scheme are pre-

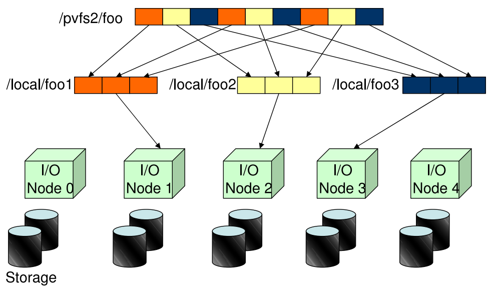


Figure 1: Striping a file over parallel disks. Striping is performed at two levels, the first of which can be exposed to and controlled by software.

sented in Section 6. Section 7 discusses related work and Section 8 concludes the paper by summarizing its main contributions and discussing briefly possible future extensions.

2 Disk System Architecture and Importance of Disk Reuse

Figure 1 depicts the disk system architecture targeted by our work. In many high-performance storage systems today, there are two levels of *striping*. The first one, which is at the software level, divides an array into equal-sized blocks (stripes) and distributes these blocks across a number of I/O nodes in a round-robin fashion. The second level of striping occurs at an I/O node level where the data blocks mapped to an I/O node are further striped (at a much finer granularity) over the disks managed by that I/O node (e.g., using one of the RAID schemes [4]). While this second level of striping is not visible to the software, the first level of striping is; and in fact, many modern file systems provide hints that can be used to query or control some of the striping parameters (e.g., the number of I/O nodes to be used for striping data, the I/O node from which the striping begins, and the size of a stripe). The compiler-based disk reuse optimization approach presented in this work focuses on this software-level striping. In our discussion, we assume a single disk per I/O node, and therefore, we use the terms “I/O node” and “disk” interchangeably as long as the context is clear.

We also assume that a portion of the main memory of the computation node is reserved to serve as *buffer* (also called *cache*) for frequently used disk data. If a requested data item is found in this cache, no disk I/O is performed and this can reduce data access latencies significantly. While it is possible to employ several buffer management schemes, the one used in this work operates under the LRU policy which replaces the least recently used stripe when a new block is to be brought in. Selection of the buffer management scheme to employ is orthogonal to the main focus of this paper. It is important to note that all the disks in the system share the same buffer (in the computation node) to cache their data, and thus, effective management of this buffer is very critical. Note also that, in addition to the cache in the computation node, the I/O nodes themselves can also employ caches. Our optimization target in this paper, however, is the performance of the cache in the computation node.

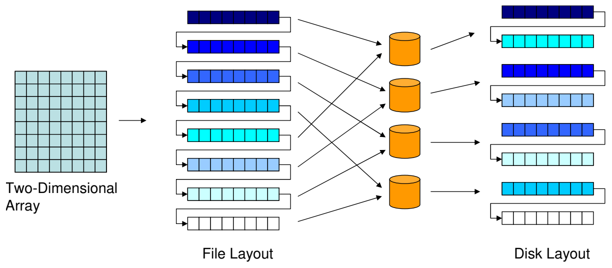


Figure 2: Different mappings a two-dimensional disk-resident array goes through. In the most general case, the memory layout, file layout and disk layout for an array can be all different from each other. We use D to represent a mapping from file layout to disk layout.

Figure 2 shows the *mappings* a two-dimensional data array data goes through as far as disk system storage is concerned. Array data in memory is stored in file using some storage order, which may be row-major, column-major, or in a blocked fashion. This is called the *file layout*. (Note that this may be different from the memory layout adopted by the underlying programming language. For example, a C array can be stored in file using column-major layout as opposed to row-major, which is the default memory layout for multi-dimensional arrays in C.) The file is then striped across the available disks on the system. Therefore, two data elements which are neighbors in the memory space can get mapped to separate disks as a result of this series of mappings. Similarly, data blocks that are far apart from each other can get mapped to the same disk as a result of striping. In this work, when we use the term “disk-resident array,” we mean an array that is mapped to the storage system using these mappings. Note that, while it is also possible to map multiple data arrays to one file or one data array to multiple files, in this work we consider only one-to-one mappings between data arrays and files. However, our approach can easily be extended, if desired, to work with one-to-many or many-to-one mappings as well. Unless otherwise stated, all the data arrays mentioned in this paper are disk resident.

Let us now discuss why disk reuse is important and how an optimizing compiler can improve it. The next section gives the technical details of our proposed compiler-based approach to improving disk reuse. Recall that disk reuse means using a data in a given set of

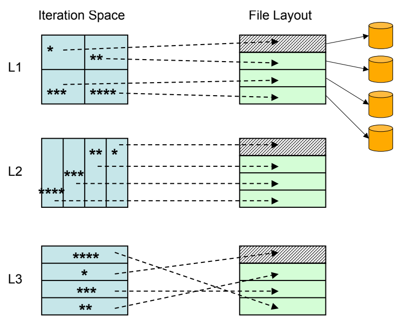


Figure 3: An example access patterns and the corresponding file layout. We restructure the application code such that, once a particular file block (stripe) is accessed, iterations that access the same block extracted from all loop nests (L1, L2, and L3) are executed together. Note that all three loop nests access the same disk-resident array.

```
/* Open files, U, V, and W */
MPI_File_open (···, U, &fh_U, ···);
MPI_File_open (···, V, &fh_V, ···);
MPI_File_open (···, W, &fh_W, ···);
/* Loop on horizontal file block */
for ii=1, R, 1 {
 /* Read next block of matrix U */
 MPI_File_read (fh_U, ···);
 /* Loop on vertical file block */
 for jj=1, R, 1 {
  /* Read next block of matrix V */
  MPI_File_read (fh_V, ···);
  /* actual matrix product */
  for i=1, N, 1
   for j=1, N, 1
    for k=1, N, 1
     W[i, j] += U[i, k] * V[k, j];
  /* Write block of W */
  MPI_File_write (fh_W, ···);
 }
}
/* close all open files */
MPI_File_close( &fh_U );
MPI_File_close( &fh_V );
MPI_File_close( &fh_W );
```

(a)

```
for i=1, N × R, 1
 for j=1, N × R, 1
  for k=1, N × R, 1
   W[i, j] += U[i, k] * V[k, j];
```

(b)

Figure 4: (a) A matrix multiplication code written in MPI-IO that operates on disk resident arrays. In this example code, each file is divided into $R \times R$ blocks and each block has $N \times N$ elements. (b) The corresponding simplified version that omits the file I/O commands and highlights computation.

disks as much as possible before accessing other disks.

• When disk reuse is improved, the chances of finding the requested data in the buffer increases. As a simple scenario, consider a case in which a given disk resident array is divided into four stripes and each stripe is stored on a separate disk. We can expect a very good buffer performance if the code can be restructured such that accesses to a given disk are clustered together. This is because such a clustering improves chances for catching data in the buffer at the time of reuse. Figure 3 illustrates this scenario. In this scenario, three different loop nests (L1, L2, and L3) access a given disk-resident array. Figure 3 shows which portions of the iteration spaces of these loop nests access what stripes (we assume 4 stripes). In a default execution, the iteration space can be traversed in a row-wise fashion. As a result, for example, when the first row of L1 is executed, two stripes are accessed (and they compete for the same buffer in the computation node). In our approach however the iteration spaces are visited in a buffer-aware fashion. If dependencies allow, we first execute the chunks (marked using *) from L1, L2, and L3 (one after another). Note that all these chunks (iterations) use the same stripe (and therefore achieve a very good data reuse in the buffer). After these, the chunks marked ** are executed, and these are followed by the chunks marked ***, and so on.

• Since our approach clusters disk accesses to a small set of disks at any given time (and maximizes the number of unused disks), in a storage system that accommodates power-saving features, unused disks can be placed into a low-power operating mode [25, 30, 40]. However, in this paper we do not quantify the power benefits of our approach.

As mentioned earlier, our work focuses on I/O intensive applications that process disk resident arrays. The application codes we target are written in MPI-IO [33], which is the part of the MPI library [12] that handles file I/O related activities. MPI-IO allows synchronous and asynchronous file reads and writes as well as a large set of collective file operations. Figure 4(a) shows an MPI-IO code fragment that performs matrix multiplication on disk resident arrays. For clarity reasons, in our discussion, we omit the MPI-IO commands and represent such a code as shown in Figure 4(b). That is, all the code fragments discussed in this paper are assumed to have the corresponding file I/O commands.

3 Mathematical Details

To capture disk accesses and optimize them, we use polyhedral algebra based on Presburger Arithmetic. Presburger formulas are made of arithmetic and logic connectives and existential ($\exists$) and universal ($\forall$) quantifiers. In our context, we used them to capture and enumerate loop iterations that exhibit disk access locality. We use the term *disk map* to capture a particular set of disks (I/O nodes) in the system. For a storage system with T disks, we use $\Lambda = \lambda_1\lambda_2\lambda_3 \cdots \lambda_T$ to indicate a disk map. As an example, if $T = 4$, $\lambda_1\lambda_2\lambda_3\lambda_4 = 0110$ represents a subset (of disks) that includes only the second and third disks, whereas 1110 specifies a subset that includes all disks

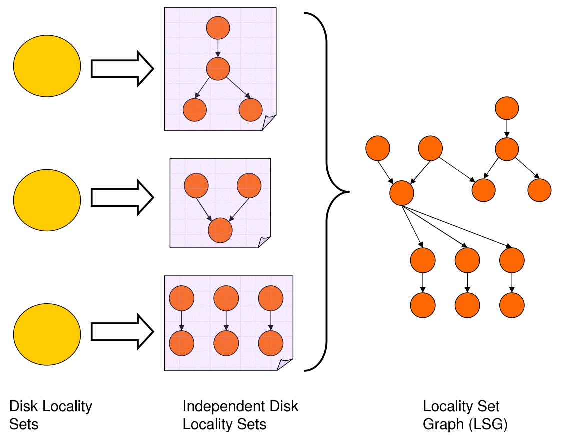


Figure 5: An example that shows three disk locality sets, their independent locality sets, and an LSG. Each independent LSG in the middle corresponds each disk locality set on the left. Edges in the independent LSGs represent the dependencies between the nodes. The LSG on the right is generated by combining each independent LSG taking accounts of dependencies.

except the last one. Assuming that array–to–disk mapping (e.g., such as one shown in Figure 2) is exposed to the compiler, the compiler can set up a relationship between the loop iterations in an application and the disks in the storage system the corresponding (accessed) array elements are stored. For example, if array reference $U[i+1]$ appears in a loop with iterator i, for a given value of i we can determine the disk that stores the corresponding array element ($U[i+1]$).

We define *disk locality set* as a set of loop iterations – which may belong to any loop nest in the application code – that access the set of disks represented by the same disk map. Mathematically, for a given disk map Λ, we can define the corresponding disk locality set ($\mathcal{Q}_\Lambda$) as:

$$\mathcal{Q}_\Lambda = \{\vec{\xi} \,|\, \vec{\xi} \in \mathcal{I} \wedge \{\exists R \in \mathcal{R} \text{ such that } D(R(\vec{\xi})) \in \Lambda\}\},$$

where $\mathcal{I}$ represents the set of all loop iterations (coming from all nests) in the program; $\vec{\xi}$ is a particular loop iteration; $\mathcal{R}$ represents the set of all references to disk-resident arrays; $R(.)$ is a reference (a mapping from the loop iterations to the data elements), and $D(.)$ is a disk mapping (striping) function, which maps the data elements to the disks in the system. We use expression $D(R(\vec{\xi})) \in \Lambda$ to indicate that the data element accessed via $R(\vec{\xi})$ is mapped to one of disks in the set represented by disk map Λ.

Let us give an example to illustrate the disk locality set concept. Assume that a disk-resident array U of size K is striped over 4 disks with a stripe of size $K/4$ (i.e., each disk has a single stripe). Assume further that, for the sake of illustration, we have a single loop i that iterates from 1 to $K-2$ and uses two references, $U[i]$ and $U[i+2]$, to access this disk-resident array. In this case, we have:

$$\begin{aligned}\mathcal{Q}_{1100} &= \{\xi \;\;|\;\; [1 \leq \xi \leq K-2] \\ &\quad \wedge\{[1 \leq \xi \leq K/2] \vee [1 \leq \xi+2 \leq K/2]\}\} \\ &= \{\xi \;\;|\;\; 1 \leq \xi \leq K/2-2\},\end{aligned}$$

which gives us the set of iterations that access only the first two disks. (Note that, the $\{[1 \leq \xi \leq K/2] \vee [1 \leq \xi+2 \leq K/2]\}$ part is due to two references to array U, and since the loop nest has only a single loop, we use ξ instead of $\vec{\xi}$.)

An important characteristic of the iterations that belong to the same $\mathcal{Q}_\Lambda$ is that they exhibit a certain degree of locality as far as disks are concerned. As a result, if, somehow, we can transform the application code and execute iterations that belong to the same $\mathcal{Q}_\Lambda$ successively, we can improve disk reuse (as in the case illustrated in Figure 3). However, this is not very trivial in practice because of two reasons. First, the inter-iteration data dependencies in the application code may not allow such an ordering, i.e., we may not be able to restructure the code for disk reuse and (at the same time) maintain its original semantics. Second, even if such an ordering is legal from the viewpoint of data dependencies, it is not clear how it can be obtained, i.e., what type of code restructuring can be applied to obtain the desired ordering. More specifically, it is not clear whether the transformation (code restructuring) requested for clustering accesses to a subset of disks can be obtained using a combination of well-known transformations such as loop fusion, loop permutation, and iteration space tiling [36]. From a compiler angle, there is nothing much to do for the first reason. But, for the second one, polyhedral algebra can be of help, which is investigated in the rest of the paper.

Suppose, for now, that the application code we have has no dependencies (we will drop this assumption shortly). In this case, we may be able to improve disk reuse (and the performance of the buffer in the computation node) using the following two-step procedure:

• For any given Λ, execute iterations in the $\mathcal{Q}_\Lambda$ set consecutively, and

• In moving from $\mathcal{Q}_\Lambda$ to $\mathcal{Q}_{\Lambda'}$, select Λ' such that the Hamming Distance between Λ and Λ' is minimum when all possible Λ's are considered.

The first item above helps us have good disk reuse by executing the iterations that belong to the subset of disks represented by a given disk map. The second item, on the other hand, helps us minimize the number of disks whose status (i.e., being used or not being used) has to be changed as we move from executing the iterations in $\mathcal{Q}_\Lambda$

to executing the iterations in $\mathcal{Q}_{\Lambda'}$. As a result, by applying these two rules repeatedly, one can traverse the entire iteration space in a disk-reuse efficient manner, and this in turn helps improve the performance of the buffer.

However, real I/O-intensive applications typically have lots of data dependencies and, thus, the simple approach explained above will not suffice in practice. We now discuss how the compiler can capture the dependencies that occur across the different disk locality sets.

We start by observing that the iterations in a given disk locality set $\mathcal{Q}_\Lambda$ can have data dependencies amongst themselves. We consider a partitioning (such a partitioning can be obtained using the Omega library [26] or similar polyhedral tools) of $\mathcal{Q}_\Lambda$ into subsets $\mathcal{Q}_{\Lambda,1}$, $\mathcal{Q}_{\Lambda,2}$, $\cdots$, $\mathcal{Q}_{\Lambda,s}$ such that $\mathcal{Q}_{\Lambda,i} \cap \mathcal{Q}_{\Lambda,j} = \emptyset$ for any i and j, $\mathcal{Q}_{\Lambda,1} \cup \mathcal{Q}_{\Lambda,2} \cup \cdots \cup \mathcal{Q}_{\Lambda,s} = \mathcal{Q}_\Lambda$, and for any i and j, all data dependencies are either from $\mathcal{Q}_{\Lambda,i}$ to $\mathcal{Q}_{\Lambda,j}$ or from $\mathcal{Q}_{\Lambda,j}$ to $\mathcal{Q}_{\Lambda,i}$. The first two of these constraints indicate that the subsets are disjoint and collectively cover all the iterations in $\mathcal{Q}_\Lambda$, and the last constraint specifies that, as far as $\mathcal{Q}_\Lambda$ is concerned, the iterations in any $\mathcal{Q}_{\Lambda,i}$ can be executed successively without any need of executing an iteration from the set $\mathcal{Q}_\Lambda - \mathcal{Q}_{\Lambda,i}$. That is, when we start executing the first iteration in $\mathcal{Q}_{\Lambda,i}$, all the remaining iterations in $\mathcal{Q}_{\Lambda,i}$ can be executed one after another (of course, these iterations can have dependencies among themselves). We refer to any such subset $\mathcal{Q}_{\Lambda,i}$ of $\mathcal{Q}_\Lambda$ as the "independent disk locality set," or the "independent set" for short. As an example, Figure 5 shows three locality sets (on the left) and the corresponding independent locality sets (in the middle). The first locality set in this example contains four independent locality sets, and these independent locality sets are connected to each other using three dependence edges. In our approach, independent locality sets are the building blocks for the main – graph based – data structure used by the compiler for disk reuse optimization.

This graph, called the "locality set graph" or LSG for short, can be defined as LSG=(V, E) where each element of V represents an independent disk locality set, and the edges in E capture the dependencies between the elements of V. In other words, an LSG has the $\mathcal{Q}_{\Lambda,i}$ sets as its nodes and the dependencies among them as its edges. The right portion of Figure 5 shows an example LSG. The question then is to schedule the nodes of the LSG while preserving the data dependency constraints between the nodes. What we mean by "scheduling" in this context is determining an order at which the nodes of the graph will be visited (during execution). Clearly, we want to determine such a schedule at compile time and execute it at runtime, and the goal of this scheduling should be minimizing the Hamming Distance as we move from one independent set to another. For the example LSG in Figure 5, we show in Figure 6 two legal

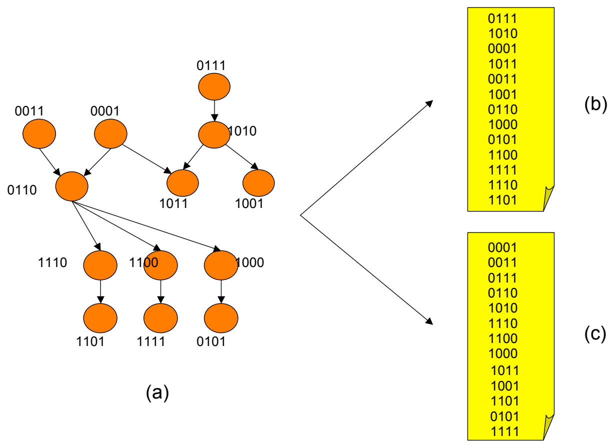


Figure 6: An example LSG (a) and two legal schedules (b and c). The order of the schedule (c) is determined based on minimum Hamming Distance, thereby exhibiting less number of disk state changes.

(dependence preserving) schedules. Note that the total (across all time steps) Hamming Distance for the first schedule (in Figure 6(b)) is 28, whereas that for the second one (in Figure 6(c)) is 15. Therefore, we can expect the second one to result in a better disk buffer reuse than the first one.

However, we note that a given LSG may *not* always be schedulable as it is. This is because it can have cycles involving a subset of its nodes. Consider for example the example LSG shown in Figure 7(a). This LSG has two cycles, and it is not possible to determine a schedule for it. In order to convert a non-schedulable LSG to a schedulable one, we somehow have to break all the cycles it contains. But, before explaining how this can be done, we want to discuss briefly the reasons for the cycles in an LSG. There are two reasons for cycles in a given LSG. First, for a given $\mathcal{Q}_\Lambda$, there can be a cycle formed by its independent sets (the $\mathcal{Q}_{\Lambda,i}$s) only. Second, the independent sets coming from the different disk locality sets can collectively form a cycle, i.e., two independent disk locality sets such as $\mathcal{Q}_{\Lambda,i}$ and $\mathcal{Q}_{\Lambda',j}$ can involve in the same cycle, where $\Lambda \neq \Lambda'$.

If an LSG has one or more cycles, we need to find a way of eliminating those cycles before the LSG can be scheduled for improving disk reuse. In the rest of this section, we discuss our solution to this issue. It can be observed that there are at least two ways of removing a cycle from a cyclic LSG. First, the nodes that are involved in the cycle can be combined into a single node. This technique is called *node merging* in this paper, and is illustrated in Figures 7(b) and (c), for the cyclic LSG in Figure 7(a). Note that, when the nodes are merged, the iterations in the combined node can be executed in an or-

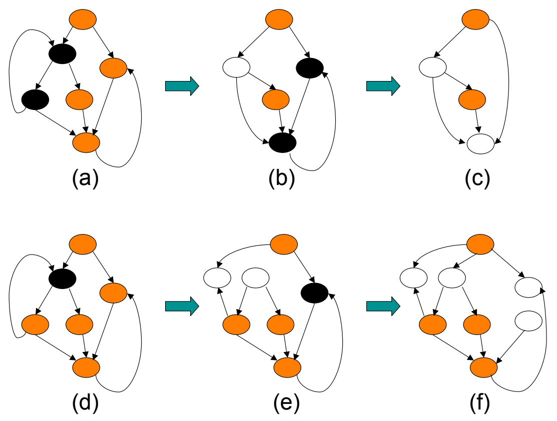


Figure 7: Application of node merging (a through c) and that of node splitting (d through f). In each case, the nodes selected for applying node merging and splitting are shown in black color, while the affected nodes are shown in white color.

der that respect data dependencies. The second technique that can be used for breaking cycles is *node splitting* (see Figures 7(d) through (f)). While these techniques can be used to convert a non-schedulable LSG to a schedulable one, each has a potential problem we need to be aware of. (clearly, one can also use a combination of node merging and node splitting to remove cycles.) A consequence of node merging is that the corresponding iteration execution may not be very good, as the two successively executed iterations (from the combined node) can access different set of disks. (note that the disk map of the merged node is the bitwise-OR of the disk maps of the involved nodes.) That is, the potential cost of eliminating cycles is a degradation in disk reuse. Node splitting on the other hand has a different problem. It needs to be noted that not all splittings can help us eliminate the cycles in a given LSG. In other words, one needs to be careful in deciding which iterations to place into each of the resulting sub-nodes so that the cyclic dependence at hand can be broken. Determination of these iterations may not be very trivial in practice, but is doable using automated compiler analysis supported by a polyhedral tool. Also, after splitting, the disk maps of the resulting nodes can be determined based on the loop iterations they contain. It is to be observed that node splitting in general also increases the code size as we typically need a separate nest for each node in the LSG.

Our preliminary experience with these two techniques showed that in general node splitting is preferable over node merging, mainly because the latter can lead to significant losses in disk reuse, depending on the application code being optimized. Therefore, in our analysis below, we restrict our discussion to node splitting only. However, as mentioned above, code size can be an issue with node splitting, and hence, we keep the number of splittings at minimum. So, the problem now becomes one of *determining the minimum number of nodes to split that makes the LSG cycle free*. We start by noting that splitting a node, if done successfully, has the same effect as that of *removing* a node (and the arrows incident on it) from the graph. That is, as far as removing cyclic dependencies is concerned, node splitting and node removal are interchangeable. Fortunately, this latter problem (removing the minimum number of nodes from a graph to make it cycle free) has been studied in the past extensively and is known as the "feedback vertex set" problem [8]. Karp was the first one to show that this problem is NP-complete on directed graphs; but it is known today that the undirected version is also NP-complete. Moreover, the problem remains NP-complete for directed graphs with no in-degree or out-degree exceeding 2, and also for planar directed graphs with no in-degree or out-degree exceeding 3. Fortunately, there exist several heuristic algorithms proposed in the literature for the feedback vertex set problem. In this work, we use the heuristic discussed in [7]. Since the details of this heuristic are beyond the scope of this paper, we do not discuss them.

As an example, Figure 8(a) gives a sample LSG. Figure 8(b) highlights the node selected by the heuristic in [7], and Figure 8(c) gives the pruned LSG. Note that splitting the node identified by [7] eliminates both the cycles. Figure 8(d) on the other hand shows the LSG after node splitting, which is cycle free. It is important to note that, while node removal and splitting obviously result in different LSGs, their effects on the schedulability of a given graph are similar; that is, both of them make a given cyclic graph schedulable. In particular, the set of nodes returned by the heuristic in [7] is the set of *minimum* nodes that need to be considered for splitting (though, as explained below, in some cases we may consider more nodes for potential split). Based on this discussion, Figure 10 gives the algorithm used by our compiler for restructuring a given code for improving disk reuse. This algorithm starts by building the initial LSG for the input code. This LSG can contain cycles, and hence, we next invoke procedure remove_cycles(.) to obtain a cycle free LSG. While this step uses the heuristic approach in [7], it needs to do some other things as well, as explained below.

In the rest of this section, we discuss details of our node splitting strategy. Once a node is identified (using the heuristic in [7]) as a potential candidate for splitting, our approach checks whether it can be split satisfactorily. What is meant by "satisfactorily" in this context is that, although in theory we can always a split a node into two or more nodes, the one we are looking for has the properties explained below.

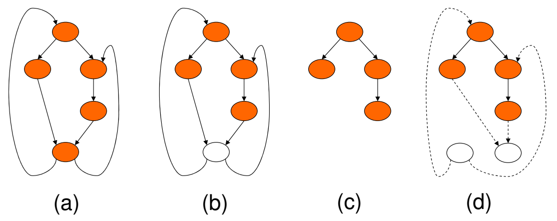


Figure 8: (a) An example LSG. (b) The node selected by the algorithm in [7] for eliminating cycles. (c) The graph after the cycles have been eliminated. (d) The graph after the node detected by the algorithm in [7] is split into two nodes.

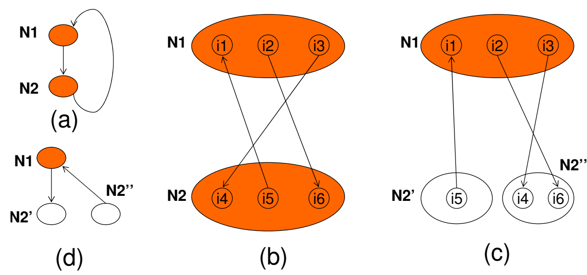


Figure 9: (a) An example LSG. (b) Details of dependencies of the LSG. The node N1 contains the iterations i1 through i3 whereas the node N2 contains the iterations i4 through i6. (c) Detailed view after splitting. (d) The graph after splitting. In (b), iterations i4 and i6 are the iterations that have incoming dependence from the node N1, the iteration i5, on the other hand, is the iteration that has outgoing dependence edge to the node N1.

Assume that $\mathcal{Q}_\Lambda$ is the node to be split. Let $\mathcal{G}_I$ be the set of nodes from which there are dependencies to node $\mathcal{Q}_\Lambda$. That is, for each member, $\mathcal{Q}_{\Lambda'}$, of $\mathcal{G}_I$, there is at least a dependence from $\mathcal{Q}_{\Lambda'}$ to $\mathcal{Q}_\Lambda$. Assume further that $\mathcal{G}_O$ is the set of nodes to which we have dependencies from node $\mathcal{Q}_\Lambda$. In other words, we have a dependence from $\mathcal{Q}_\Lambda$ to each node, $\mathcal{Q}_{\Lambda''}$, of $\mathcal{G}_O$. Suppose now that $\mathcal{Q}_\Lambda$ is split into two sub-nodes: $\mathcal{Q}_{\Lambda_a}$ and $\mathcal{Q}_{\Lambda_b}$. We call this split "satisfactory" if all of the following three conditions are satisfied after the split:

• No dependency goes from any $\mathcal{Q}_{\Lambda'} \in \mathcal{G}_I$ to $\mathcal{Q}_{\Lambda_b}$. In other words, all the original in-coming dependencies of $\mathcal{Q}_\Lambda$ are directed to $\mathcal{Q}_{\Lambda_a}$.

• No dependency goes from $\mathcal{Q}_{\Lambda_b}$ to any $\mathcal{Q}_{\Lambda''} \in \mathcal{G}_O$. In other words, all the original out-going dependencies of $\mathcal{Q}_\Lambda$ are directed from $\mathcal{Q}_{\Lambda_b}$.

• No dependency exists between $\mathcal{Q}_{\Lambda_a}$ and $\mathcal{Q}_{\Lambda_b}$.

We say that cycle in question is removed (if the removal of $\mathcal{Q}_\Lambda$ is sufficient to remove the cycle; otherwise, the other nodes in the cycle, which is detected by the algorithm in [7], have to be visited), if the three conditions above are satisfied. It needs to be emphasized that, precisely speaking, the last condition above is not always necessary. However, if there exist dependencies between $\mathcal{Q}_{\Lambda_a}$ and $\mathcal{Q}_{\Lambda_b}$, it is possible that we still have cycle(s) in the LSG due to $\mathcal{Q}_\Lambda$, depending on the direction of these dependencies. Enforcing the last condition, along with the others, ensures that the cyclic dependencies are removed completely. Figure 9 illustrates an example LSG. Figure 9(a) shows an original LSG in coarse grain and (b) illustrates the dependencies between the two nodes of this LSG in fine grain. Assuming that the bottom node has been selected for removal (ultimately one of possible nodes to be split) by the heuristic in [7], Figures 9(c) and (d) show the result of splitting LSG in fine grain and coarse grain, respectively. As an another example, Figure 8(d) shows the split version of the LSG in Figure 8(a). Note that, due to the third condition above, our approach may need to look at more nodes than ones determined by the heuristic described in [7]. We also need to mention that, in our implementation, these three conditions listed above are checked using the Omega library [26].

4 File Layout Modification

So far in our discussion we considered only data access pattern restructuring for improving disk reuse. It is to be noted however that data layout on disks can also play an important role as far as disk reuse is concerned. Specifically, a different file layout can lead to a different disk layout which can in turn lead to a different amount of disk reuse. Let us consider the following code fragment:

```
for $i$=1,$N$,1
    for $j$=1,$N$,1
        $U[i,j] = f(V[j,i]);$
```

In this code fragment, two disk-resident arrays are accessed (as mentioned earlier, we do not show explicit I/O statements). While one of these is accessed in row wise, the second one is traversed column wise. Consequently, storing both the arrays in the same fashion in file (e.g., as shown in Figure 2) may not be the best option since such a storage will not be able to take advantage of disk reuse for the second array as its data access pattern and file storage pattern would not match. Now, consider the file layout transformation depicted in Figure 11. If this transformation is applied to the second array (V) in the code fragment above, we can expect better disk reuse.

The important question to address is to select the mapping that maximizes disk reuse. We start by noting that the search space is very large for potential file layout transformations, as there are many ways of transforming a file layout. However, our experience with disk-intensive scientific applications and our preliminary experiments suggest that we can restrict the potential map-

```
$MAX$ — maximum number of node splitting operations allowed;
$\mathcal{Q}_{\Lambda}$ — disk locality sets;
$dep(i, j)$ — returns TRUE if dependence between $\mathcal{Q}_{\Lambda,i}$ and $\mathcal{Q}_{\Lambda,j}$ exists;
$H(\mathcal{Q}_{\Lambda,i}, \mathcal{Q}_{\Lambda,j})$ — returns Hamming distance between $\mathcal{Q}_{\Lambda,i}$ and $\mathcal{Q}_{\Lambda,j}$;
$VS$ — set of $\mathcal{Q}_{\Lambda,i}$s that are ready to schedule;
$\mathcal{Q}_{\Lambda,x}$ — last scheduled locality set;

procedure build_LSG() {
  for each $\mathcal{Q}_{\Lambda,i}$ {
    build independent disk locality sets;
  }
  for any two subsets $\mathcal{Q}_{\Lambda,i}$ and $\mathcal{Q}_{\Lambda,j} \in \mathcal{Q}_{\Lambda}$ {
    if ($dep(i, j)$ == TRUE) add edge between two nodes, $\mathcal{Q}_{\Lambda,i}$ and $\mathcal{Q}_{\Lambda,j}$;
  }
}

procedure remove_cycles() {
  split_count = 0;
  while (split_count < $MAX$) {
    apply node splitting;
    split_count++;
  }
}

procedure schedule_LSG() {
  $VS = \emptyset$;
  for each $\mathcal{Q}_{\Lambda,i} \in LSG$ {
    if ($\mathcal{Q}_{\Lambda,i}$ has no parents || $\forall$ parents of $\mathcal{Q}_{\Lambda,i}$ has been scheduled) {
      $VS = VS \cup \mathcal{Q}_{\Lambda,i}$;
    }
  }
  while ($VS \neq \emptyset$) {
    select $\mathcal{Q}_{\Lambda,y} \in VS$ such that $H(\mathcal{Q}_{\Lambda,x}, \mathcal{Q}_{\Lambda,y})$ is minimum;
    schedule the selected $\mathcal{Q}_{\Lambda,y}$;
    $LSG = LSG - \mathcal{Q}_{\Lambda,y}$;   /* remove $\mathcal{Q}_{\Lambda,y}$ from $LSG$; */
    $VS = VS - \mathcal{Q}_{\Lambda,y}$;     /* update $VS$ */
    set $\mathcal{Q}_{\Lambda,x}$ to $\mathcal{Q}_{\Lambda,y}$;   /* update last scheduled locality set */
  }
}

main() {
  call build_LSG();
  if (exits cycles in $LSG$)
    call remove_cycles();
  while ($LSG \neq \emptyset$) {
    call schedule_LSG();
  }
}
```

Figure 10: Compiler algorithm for scheduling a given code to increase disk reuse. Our algorithm starts by building the LSG and then removes the cycles in the graph if there are any. After obtaining cycle-free LSG, we schedule each node in the graph such that the next node scheduled has the minimum Hamming Distance from the current node. Note that, if desired, this algorithm can be applied to smaller code segments (e.g., a loop nest) as well, instead of the whole program.

pings to dimension permutations. What we mean by "dimension permutation" in this context is reindexing the dimensions of the disk resident array. As an example, restricting ourselves to dimension re-indexings, a three-dimensional disk-resident array can have 6 different file layouts. Let us use $D' = DP$ to represent the disk mapping function when file layout modification is considered, where D is the original disk mapping function discussed earlier in Section 3 and P is a permutation matrix (that implements dimension re-indexing). For example, the file layout mapping shown in Figure 11 can be expressed using the transformation matrix

$$P = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

Note that, for an m-dimensional disk-resident array, P is $m \times m$. (for example, elements on (i, j) data space is mapped to (j, i) as a result of applying the P matrix shown above to a two-dimensional array.)

It is to be noted, however, that the decision for selection of a permutation should be made carefully by considering all the statements that access the array in question. This is because different statements in the application code can access the same disk-resident array using entirely different access patterns, and a layout transformation that does not consider all of them may end up with one that is not good when considered globally. Our file layout selection algorithm is a *profile based* one. In this approach, the application code is profiled by instrumenting it and attaching a set of counters to each disk resident array. For an m-dimensional array, we have $m! + 1$ counters, each keeping the number of times a particular file layout is preferred (note that the total number of possible dimension permutations is $m!$ and one additional counter is used for representing other file layout preferences such as diagonal layouts, for which we do not perform any optimization). In this work, we implement only dimension permutation because other file layouts such as diagonal layouts or blocked layouts are hardly uniform across all execution. Therefore, we do not take any actions for such layouts. At the end of profiling, the layout preference with the largest counter value is selected as the file layout for that array. Figure 12 gives the pseudo code for our file layout selection algorithm. As an example, let us assume that there are three loop nests (with the same number of iterations) accessing the same data array stored in a file. Assume further that the profiling reveals that the first and third loop nests exhibit column-major access pattern whereas the second loop nest exhibits row-major access pattern. As the column-major file layout is preferred more (that is, it will have a higher counter value), we select the corresponding permutation matrix and convert the file layout accordingly.

5 Parallel Execution

It is also important to study disk reuse under parallel execution. An important challenge in this case is to coordinate the disk accesses coming from multiple threads. We note that, even if the disk accesses from individual threads exhibit disk reuse, this does not necessarily mean that the overall execution will have disk reuse. The example in Figure 13 shows a scenario with two threads.

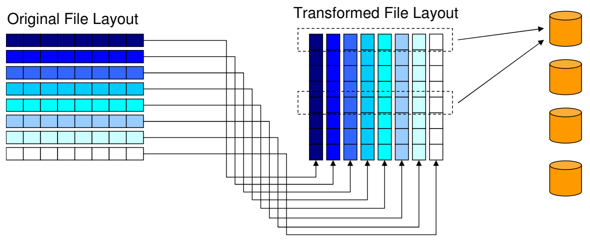


Figure 11: Converting file layout of a two-dimensional disk resident array. In this example, the original file layout in row-major order is transformed into the layout in column-major order.

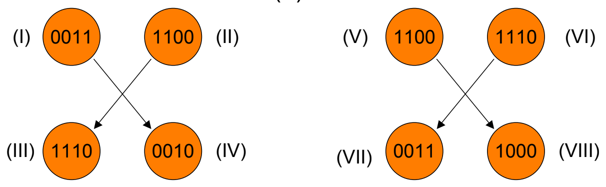
```
$N$ — number of arrays;
$U_i$ — accessed arrays, where $1 \leq i \leq N$;
$D_i$ — original disk mapping for array $U_i$;
Dim[$i$] — dimension of array $U_i$;
$P_n$ — group of permutation matrices;

for $i$ = 1 to $N$ {
    Dim[$i$] = dimension of array $U_i$;
}

C[$i$][Dim[$i$]!+1]; /* counters for each array, $U_i$, for profiling */

/* perform profiling */
for each array $U_i$ {
    for each statement accessing $U_i$ {
        detect the file layout of accessing $U_i$;
        select $P_j$ for the determined file layout;   /* $P_j$ is $j$th entry of $P_n$ */
        C[$i$][$j$]++;   /* increase corresponding counters */
    }
}

for each array $U_i$ {
    select $P_j$ that has the highest C[$i$][$j$] value;
    apply $D'_i$ = $D_i P_j$;     /* transform file layouts */
}
```

Figure 12: Compiler algorithm for transforming file layouts to improve disk reuse. Our algorithm is based on profiling that reveals the most desirable access patterns for each array across all statements within a program.

Assuming the LSGs shown in Figure 13(a), a possible scheduling is given in Figure 13(b). The overall disk reuse (when considering both the thread) in this case is not very good, though scheduling for each thread exhibits high reuse when considered alone. We now consider the alternate scheduling illustrated in Figure 13(c). In this scheduling, the overall disk reuse is very good, which is achieved by scheduling the node in individual thread such that, when both threads execute the selected nodes (at the same scheduling step), the number of disks used is minimized. Note that, within a thread, a node that is scheduled next is chosen based on the minimum Hamming Distance. By adapting this schedule, we do not have any scheduling step in Figure 13(c) that uses all four disks at the same time, whereas, in Figure 13(b), steps 1 and 4 have full usage of all disks, which is not good as far as the buffer (cache) utilization is concerned.

Our scheduling algorithm for an architecture with P

(a)



Step					
1	(I)	0011	(V)	1100	1111
2	(IV)	0010	(VI)	1110	1110
3	(II)	1100	(VIII)	1000	1100
4	(III)	1110	(VII)	0011	1111

(b)

Step					
1	(II)	1100	(V)	1100	1100
2	(III)	1110	(VI)	1110	1110
3	(I)	0011	(VIII)	1000	1011
4	(IV)	0010	(VII)	0011	0011

(c)

Figure 13: (a) LSGs for two threads of the same application. (b-c) Two legal schedules. The last columns in (b-c) are the disk status two threads, which are obtained by bitwise-ORing the disk status of two threads.

processors and D disks is given in Figure 14. This algorithm takes the LSGs as input, and determines, for each thread, the schedule of the nodes considering the global (inter-thread) usage of the disks. It uses a D-bit global variable G to represent the current usage of the disks. It schedules a node that is ready to be scheduled for each thread that finishes its current task. At each step, the algorithm first tries to schedule the node whose disk requirement can be satisfied with the set of disks currently being used. If multiple nodes satisfy this criterion, we select the one that requires the maximum number of disks to make full utilization of the currently used disks. If such a node does not exist, our algorithm schedules the node whose tag is the closest (in terms of Hamming Distance) to G, the bit pattern that represents the current disk usage (i.e., the disk usage at that particular point in scheduling). This is to minimize the number of disks whose (active/idle) states need to be changed. We want to mention that, since each node may have different execution latency it is possible that the targeted disk reuse (across threads) may not always be achieved. However, we can expect the resulting disk reuse (and buffer performance) to be better than what a random (but legal) scheduling would achieve.

Before moving to the discussion of our experimental results, we want to point out the tradeoff between disk reuse and performance. The parallel version of our approach tries to maximize the disk reuse, and this in turn tends to attract the accesses to a small set of disks. Consequently, in theory, this can lead to performance problems, as the effective disk parallelism is reduced. While

```
$P$ — number of processors;
$D$ — number of disks available;
$G$ — global disk usage map with $D$ bits;
$\mathcal{Q}_\Lambda$ — disk locality sets;
$H(\mathcal{Q}_{\Lambda,i})$ — returns Hamming distance between $\mathcal{Q}_{\Lambda,i}$ and $G$;
$VS[i]$ — set of $\mathcal{Q}_{\Lambda,i}$s that are ready to schedule in processor $i$;
$LSG[i]$ — $LSG$ for each processor $i$;

procedure schedule_LSG_P() {
  for $i$ = 1 to $P$ {     /* find schedulable $\mathcal{Q}_{\Lambda,j}$ from each processor $i$ */
    $VS[i]$ = $\emptyset$;     /* init each $VS[i]$ of processor $i$ */
    for each $\mathcal{Q}_{\Lambda,j} \in LSG(i)$ {
      if ($\mathcal{Q}_{\Lambda,j}$ has no parents || $\forall$ parents of $\mathcal{Q}_{\Lambda,j}$ has been scheduled) {
        $VS[i]$ = $VS[i] \cup \mathcal{Q}_{\Lambda,j}$;
      }
    }
  }
  /* schedule all $\mathcal{Q}_{\Lambda,j} \in VS[k]$ */
  while ($VS[k]$ != $\emptyset$) {
    for each processor $k$ {
      select $\mathcal{Q}_{\Lambda,i} \in VS[k]$ such that it requires the maximum number of
        disks to fully utilize the currently used disks, or $H(\mathcal{Q}_{\Lambda,i})$ is minimum;
      schedule the selected $\mathcal{Q}_{\Lambda,i}$;
      $LSG(k)$ = $LSG(k)$ - $\mathcal{Q}_{\Lambda,i}$; /* remove $\mathcal{Q}_{\Lambda,i}$ from $LSG(k)$; */
      $VS[k]$ = $VS[k]$ - $\mathcal{Q}_{\Lambda,i}$;
      update $G$ by bitwise-ORing with $\Lambda$ of $\mathcal{Q}_{\Lambda,i}$;
    }
  }
}
main() {
  /* build $LSG$ for a code assigned to each processor */
  for $i$ = 1 to $P$ {
    call build_LSG();
    if ($LSG$ has cycle(s)) {
      call remove_cycles();
    }
  }

  initialize $G$ by setting all bits to 0;
  while (exists ($\forall i$, $LSG[i]$ != $\emptyset$)) {
    call schedule_LSG_P();
  }
}
```

Figure 14: Compiler algorithm for scheduling parallel execution of a given code to increase disk reuse. We use the same procedures given in Figure 10 for building LSG and removing cycles.

this negative impact has already been accounted for in our experiments (discussed in the next section), we observed that its magnitude is not very high. However, this magnitude is typically a function of the application access pattern and disk system parameters as well, and further studies are needed to reach better evaluations.

6 Experiments

6.1 Setup

We implemented our compiler algorithm using the SUIF infrastructure [13]. Our disk reuse optimization increased compilation times of the original applications by about 55% on average (the largest compilation time when our optimization is applied was 87 seconds). When the file layout optimization is enabled, the largest compilation time jumped to 116 seconds. Extension for parallelism added another 9 seconds on average.

Table 1: System parameters.

Parameter	Value
CPU	
Model	Intel P4
Clock Frequency	2.6 GHz
Memory System	
Model	Rambus DRAM
Buffer Capacity	1 GB
Disk System	
Number of I/O Nodes	8
Data Striping	Uses all 8 I/O nodes
Stripe Size	64 KB
Interface	ATA
Storage Capacity/Disk	40 GB
RPM	10,000
Interconnect	
Model	Ethernet
Bandwidth	100 Mbps

We performed our experiments using a platform which includes MPI-IO [33] on top of the PVFS parallel file system [27]. PVFS is a parallel file system that stripes file data across multiple disks in different nodes in a cluster. It accommodates multiple user interfaces which include MPI-IO, traditional Linux interface, and the native PVFS library interface. In all our experiments, we used the MPI-IO interface. Table 1 gives the values of our major experiment parameters. We want to emphasize however that later we present results from our *sensitivity analysis* where we change the default values of some of the important parameters.

For each benchmark in our experimental suite, we performed experiments with different versions:

• *Base Scheme*: This represents the original code without any data locality optimization.

• *Conventional Locality Optimization* (CLO): This represents a conventional data locality optimization technique that employs loop restructuring. It is not designed for I/O, and does not take disk layout into account. The specific data reuse optimizations used include loop interchange, loop fusion, iteration tiling and unrolling. This version in a sense represents the state-of-the-art as far as data locality optimization is concerned.

• *Disk Reuse Optimization – Loop Based* (DRO-L): This is our approach applied at a loop nest level; i.e., each loop nest is optimized in isolation.

• *Disk Reuse Optimization – Whole Program Based* (DRO-WP): This is our approach applied at a whole program level.

All the versions use the MPI-IO interface [33] of PVFS [27] for performing disk I/O. Note that both DRO-L and DRO-WP are the different versions of our approach, and CLO represents the state-of-the-art as far as optimizing data locality is concerned. The reason that we make experiments with the DRO-L and DRO-WP versions separately is to see how much additional benefits one can obtain by going beyond the loop nest level

Table 2: Our applications.

Application Name	Brief Description	Data Set Size (GB)	Disk I/O Time (sec)	Total Time (sec)
sar	Synthetic Aperture Radar Kernel	21.1	64.6	101.4
hf	Hartree-Fock Method	53.6	98.3	173.6
apsi	Pollutant Distribution Modeling	49.9	101.2	238.7
wupwise	Physics/Quantum Chromo-dynamics	27.9	270.3	404.7
e_elem	Finite Element Electromagnetic Modeling	66.2	99.2	191.9
astro	Analysis of Astronomical Data	58.3	171.6	276.4
contour	Contour Displaying	58.7	198.6	338.4

in optimizing for I/O. In addition to these versions, we also implemented and conducted experiments with the file layout optimization scheme discussed in Section 4 and with the parallel version of our approach explained in Section 5.

The unit of buffer (cache) management in our implementation is a data block, and its size is the same as that of a stripe. The set of applications used in this study is given in Table 2. These applications are collected from different sources and their common characteristic is that they are disk-intensive. apsi and wupwise are similar to their Spec2000 counterparts [14], except that they operate on disk-resident data. The second column briefly explains each benchmark, and the third column gives the total (disk resident) data set size processed by each application. The fourth and fifth columns give the disk I/O times and total execution times, respectively, under the *base scheme* explained above. Note that both the base version and the CLO version are already optimized for buffer usage. In addition, the CLO version is optimized for data locality using conventional techniques, as explained above. Therefore, the performance improvements brought by our approaches (DRO-L and DRO-WP) over these schemes (base and CLO) are due to the code re-ordering we apply. We also see from Table 2 that the contribution of disk I/O times to overall execution times varies between 42.4% and 63.7%, averaging on 57.4%. Therefore, reducing disk I/O times can be very useful in practice. In the remainder of this section, we present and discuss the performance improvements brought by our approach. The disks I/O time savings and overall execution time savings presented below are with respect to the fourth and fifth columns of Table 2, respectively.

6.2 Results

We start by presenting the percentage improvements in disk I/O times brought by the three optimized versions explained above. The results shown in Figure 15 indicate that the average improvements brought by CLO, DRO-L and DRO-WP over the base scheme are 10.4%, 16.1% and 23.9%, respectively. Overall, we see that, while DRO-L performs better than CLO, by 6.3% on average, the best savings are obtained – for all benchmarks tested – with the DRO-WP version, on average, 15.0% and 9.3% over CLO and DRO-L, respectively, meaning that going beyond a single loop nest is important in maximizing the buffer performance. While these improvements in disk I/O times are important, we also need to look at the savings in overall execution times, which include the computation times as well. These results are presented in Figure 16 and show that the average improvements with the CLO, DRO-L and DRO-WP versions are 5.9%, 9.0% and 13.5%, respectively. To better explain how our approach achieves much more performance improvements over conventional data reuse optimization, we present in Figure 17 the average number of times a given data block is visited under the different schemes (that is, how many times a given data block (on average) is brought from disks to cache). We observe that this number is much lower with the DRO-WP version, explaining the additional performance benefits it brings. In fact, on average, the number of disk traversals per block is 3.9 and 2.1 with the base version and DRO-WP, respectively.

Sensitivity Analysis. In this section, we study the sensitivity of our performance savings to several parameters. A critical parameter of interest is the buffer (cache) size. Recall that the default buffer size used in our experimental evaluation so far was 1GB. The graph in Figure 18 gives the results using different buffer sizes. Each point in this graph represents the average value (performance improvement in overall execution time), for a given version, when all seven benchmarks are considered. As expected, the performance gains brought by our approach get reduced when increasing the buffer size. However, even with the largest buffer size we used, the average improvement we have (with the DRO-WP version) is about 6.7%, underlining the importance of disk reuse optimization for better performance. Considering the fact that data set sizes of disk-intensive applications keep continuously increasing, one can expect the disk reuse based approach to be more effective in the future. To elaborate on this issue further, we also performed experiments with larger data sets. Recall that the third column of Table 2 gives the data set sizes used in our experiments so far. Figure 19 gives the average performance improvements, for 1GB and 4GB buffer sizes and two sets of inputs. SMALL refers to the default dataset sizes given in Table 2, and LARGE refers to larger datasets, which are 38.2GB, 66.3GB, 82.1GB, 38.0GB, 88.1GB, 73.7GB, and 81.8GB for sar, hf, apsi, wupwise, e_elem, astro and contour, respectively. We see that our approach performs better with larger data set sizes. This is because a larger data set puts more pressure on the buffer, which makes effective utilization of buffer even more critical.

The next parameter we study is the stripe size, which can also be changed using a PVFS call when creating the file. The performance improvement results with different stripe sizes are presented in Figure 20. Our observation is

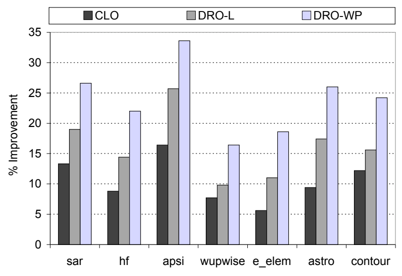


Figure 15: Performance improvements in I/O times.

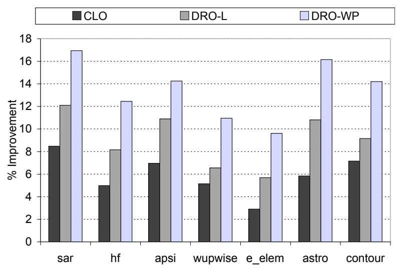


Figure 16: Performance improvements in overall execution times.

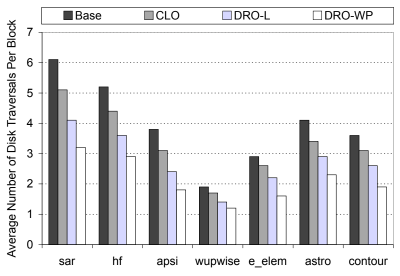


Figure 17: Average number of fetches per block. Each bar represents how many times a given block is brought to buffer cache.

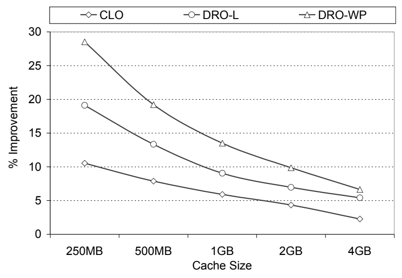


Figure 18: Sensitivity to the buffer size.

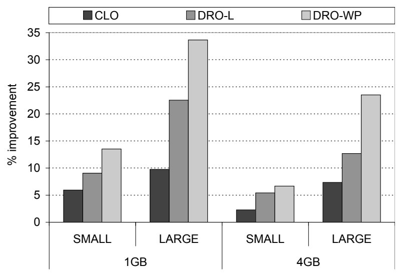


Figure 19: Sensitivity to the input size.

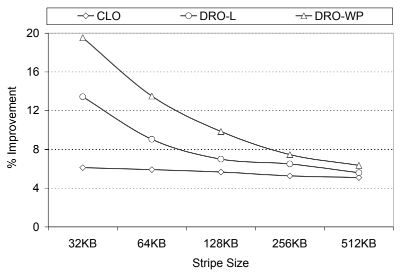


Figure 20: Sensitivity to the stripe size.

that the DRO-WP version generates the best results with all stripe sizes tested. We also see that the performance savings are higher with smaller stripe sizes. This can be explained as follows. The disk reuse is not very good in the original codes (the base scheme) with smaller stripe sizes, and since the savings shown are normalized with respect to the original codes, we observe large savings.

Comparison with I/O Prefetching. We next compare our approach to prefetching. The specific prefetch implementation we use is inspired by TIP [24], a hint-based I/O prefetching scheme. The graph in Figure 21 presents, for each benchmark, three results: prefetching alone, DRO-WP alone, and the two technique combined. We see from these results that the best performance improvements are achieved using both prefetching and code restructuring. This is because these two optimization techniques are in a sense orthogonal to each other. Specifically, while core restructuring tries to *reduce I/O latencies* by improving buffer performance, prefetching tries to *hide I/O latencies.* While it is also possible to integrate these two optimizations better (rather than applying one after another), we postpone exploring this option to a future study.

Impact of File Layout Optimization. Recall from Section 4 that file layout optimization (which impacts the layout of data on the disks as well) can help our approach improve disk reuse further. To quantify this, we performed another set of experiments, whose results are presented in Figure 22 when the whole program is optimized. We see that, except for one benchmark, layout optimization improves the effectiveness of our code restructuring approach. The average *additional* improvement it brings is about 7%. We observe that the file layout optimization could not find much opportunity for improvement in benchmark e_elem as the default file layouts of the disk resident arrays in this benchmark perform very well.

Evaluation of Parallel Execution. Figure 23 presents the results collected from an evaluation of the parallel version of our approach discussed in Section 5. For these experiments, the number of CPUs that are used to execute an application is varied between 1 and 8. For each processor size, we present the results with our baseline implementation (where disk reuse is optimized from each CPU's perspective individually) as well as with those obtained when the approach in Section 5 is enabled. We see from these results that considering all parallel threads together is important in maximizing overall disk reuse, especially with the large number of CPUs. For example, when 8 CPUs are used for executing an application, the average performance improvements with individual reuse optimization (sequential version) and collective reuse optimization (parallel version) are 25.7% and 33.3%, respectively.

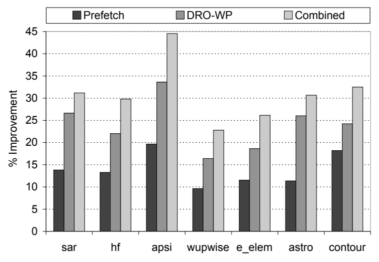


Figure 21: Comparison with I/O prefetching.

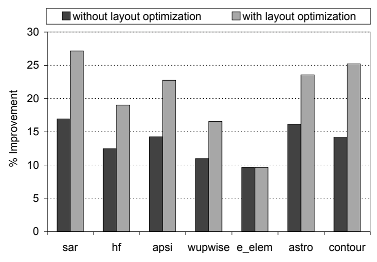


Figure 22: Impact of layout optimization.

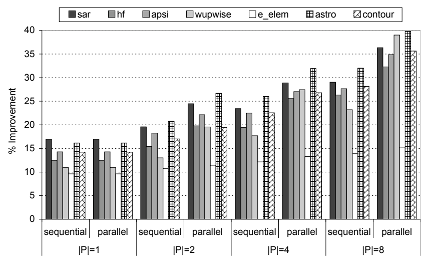


Figure 23: Impact of parallel thread optimization.

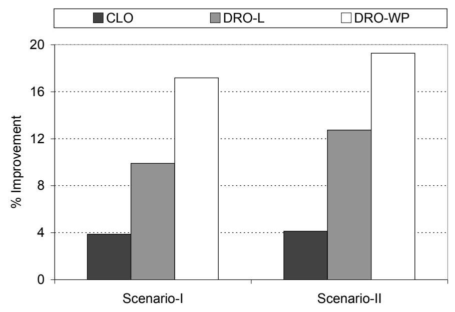


Figure 24: Impact of multiple application execution.

Evaluation of Multiple Application Execution. Since a disk system can be used by multiple applications at the same time, it is also important to quantify the benefits brought by our approach under such an execution scenario. Figure 24 presents the results from two sets of experiments. In the first set, called Scenario-I, 7 CPUs are used and each CPU executes one of our seven applications. In the second set, called Scenario-II, 8 CPUs are used and four of our applications (sar, hf, apsi and astro) are parallelized, each using two CPUs. In both set of experiments, a CPU executes only a single thread. Also, in Scenario-I the sequential version of our approach is used (for each application), and in Scenario-II the parallel version explained in Section 5 is used. The results given in Figure 24 indicate that the DRO-WP version generates the best results for both the scenarios tested.

7 Discussion of Related Work

An important way of using compiler in improving I/O performance is to hide I/O latency through I/O prefetching. Mowry et al [21] proposed a compiler-managed I/O prefetching technique for out-of-core applications. They automated the insertion of I/O prefetching instructions based on future memory page usage determined by compile-time locality analysis. In another study [2], they proposed a runtime system for managing the dynamic behavior of compiler-inserted I/O prefetch/release hints from multiple applications running concurrently. Several researches focused on compilation of I/O-intensive applications. For example, Bordawekar et al [1] focused on stencil computations and proposed several algorithms to optimize communication and ultimately to enhance I/O performance. Palecznу et al [23] proposed a technique to guide I/O for out-of-core applications based on high-level annotations, which they incorporated into Fortran D compiler. To bridge the disparities between the data access patterns and the storage (file) layouts, [18] and [17] proposed compiler-directed I/O optimization strategies. Their main idea is to find the most preferable I/O access patterns to disk-resident files, and then determine the most suitable storage layouts associated with them. For the remaining part, which contains non-dominant access patterns, they optimized it using collective I/O operations. In [35], Vilayannur et al proposed a compiler-directed discretionary caching policies for I/O-intensive applications. They leveraged compiler support in determining the cache blocks to be accessed in each loop nest. Besides these efforts to enhance the I/O performance explained so far, the compiler-guided information can be used for different purposes, such as reducing energy consumption of disk subsystems. In [30], Son et al proposed to expose disk layouts of each data file to the compiler and let the compiler analyze the data access patterns along with this information to determine I/O (disk) access patterns. These extracted disk access patterns are finally used for transforming code and/or underlying disk layouts to reduce the energy consumption of disk subsystems.

There has been significant past work on optimizing file and buffer cache management in storage systems [3, 9, 10, 16, 28, 35]. To better exploit multi-level caches, which are common in modern storage systems, several multi-level buffer cache management policies have been proposed [6, 37, 35]. [37] introduced a DEMOTE operation that allows one to keep cache blocks in an exclusive manner, i.e., a cache block is not duplicated across cache hierarchy. Chen et al [6], on the other hand, utilize the eviction information of higher level cache in deciding which cache blocks need to be replaced in a lower level of the cache hierarchy. More recently, [38] proposed a replacement policy for multi-level cache, called Karma. Karma uses application hints in maintaining cache blocks exclusively. Most high-end parallel and cluster systems provide some sort of parallel I/O operations to meet the I/O requirements (i.e., low latency and high bandwidth) of scientific applications. This is typically accomplished by employing a set of I/O nodes, each of which is equipped with multiple disks, dividing a file into a number of small file stripes, and distributing those stripes across available I/O nodes. This notion of file-level striping is adapted in many commercial or research parallel file systems, such as IBM GPFS [28], Intel PFS [11], PPFS [15], Galley [22], and PVFS2 [27]. It should be mentioned that these parallel file systems provide huge I/O performance improvements when they receive large and contiguous I/O requests. However, many scientific applications that exhibit small and non-contiguous I/O access patterns may suffer from performance degradation. To deal with this problem, several approaches have been proposed in the context of different parallel file system libraries and APIs such as Panda [5], PASSION [31], and MPI-IO [32, 33, 34]. Among various techniques used to achieve this goal, collective I/O is commonly recognized as an efficient way of reducing I/O latency. The concept of collective I/O can be implemented in different places of parallel I/O systems; namely, client side [34], disk side [19], or server side [29]. The majority of the existing collective I/O implementations employ two-phase I/O [34]. In two-phase I/O, disk accesses are reorganized in client side (compute node) before sending them over the I/O nodes. Disk-directed I/O [19], on the other hand, performs collective I/O operations on disk side, where I/O requests are optimized such that they conform to the storage layouts. In Panda [29], I/O server nodes, rather than disk or client nodes, generate I/O requests that conform to the layouts of disk-resident array data.

Our approach is different from prior compiler-based I/O optimization techniques in that it optimizes entire program rather than individual, parallel loop nests. It is also different from previous studies that considered buffer caching and prefetching because we improve I/O performance by increasing disk reuse. Lastly, our approach emphasizes the role of file layout optimization and parallel execution when applying optimization techniques to achieve better disk reuse (and better cache performance).

8 Concluding Remarks and Future Work

In the recent past, sensor, measurement, and simulation-based applications in science and engineering have grown dramatically in complexity. Moreover, there have been huge increases in the sizes of the data sets they produce, manipulate, and consume, meaning that the high I/O performance is a must for these applications. Unfortunately, advances in I/O architectures (in particular, disks) could not meet this high I/O performance requirement satisfactorily. As a result, adequate software support for I/O is critical and has to be provided at different layers, including libraries, file systems, runtime systems, and compilers. The main contribution of this paper is a compiler-directed disk performance optimization scheme for large-scale data-intensive applications. This proposed scheme is oriented towards maximizing disk reuse over successive visits to the disk system within a given period of time, thereby (1) maximizing the utilization of cache in the computation node, and (2) reducing the latencies due to data search on disks. In addition, the success of this scheme can be increased significantly if it is augmented with a file layout optimization scheme, and it can be easily adapted to capture disk interactions across the threads of a parallel application. We implemented this scheme fully using an optimizing compiler framework and evaluated its performance using seven data-intensive applications that exercise disks. The results collected indicate that our compiler-directed approach is very successful in maximizing disk reuse, and this in turn results in large savings in I/O latencies. In our experiments, we also compared our approach to a conventional data reuse optimization scheme (not designed for I/O) and explain where the additional benefits are coming from. This work shows how an optimizing compiler can help reduce I/O latencies by automated code restructuring. We believe that further compiler optimizations are possible by exposing the disk layout of data to the different layers of the software stack. One of the research directions to investigate is this interaction between the compiler optimizations for I/O and other I/O optimizations that are normally applied by file systems and runtime libraries. Another interesting research direction is to adapting the cache policy based on the application behavior information collected by the compiler. This can help to increase the hit rates of the cache, thereby further boosting the performance of the application.

Acknowledgments This work is supported in part by the NSF grants #0406340, #0444158, and #0621402. We would like to thank our anonymous reviewers for their helpful comments.

References

[1] BORDAWEKAR, R., CHOUDHARY, A. N., AND RAMANUJAM, J. Automatic Optimization of Communication in Compiling Out-of-Core Stencil Codes. In *International Conference on Supercomputing* (1996), pp. 366–373.

[2] BROWN, A. D., AND MOWRY, T. C. Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory

Intelligently. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation* (2000), pp. 31–44.

[3] Cao, P., Felten, E. W., and Li, K. Application-Controlled File Caching Policies. In *Proceedings of the USENIX Technical Conference* (1994), pp. 11–11.

[4] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A. RAID: High-Performance, Reliable Secondary Storage. *ACM Comput. Surv. 26*, 2 (1994), 145–185.

[5] Chen, Y., Winslett, M., Cho, Y., and Kuo, S. Automatic Parallel I/O Performance Optimization in Panda. In *Proceedings of the 10h Annual ACM Symposium on Parallel Algorithms and Architectures* (1998), pp. 108–118.

[6] Chen, Z., Zhou, Y., and Li, K. Eviction-based Cache Placement for Storage Caches. In *USENIX Annual Technical Conference* (2003), pp. 269–281.

[7] cheng Cai, M., Deng, X., and Zang, W. A TDI System and Its Application to Approximation Algorithms. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science* (1998), pp. 227–231.

[8] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.

[9] Cortes, T., Girona, S., and Labarta, J. Design Issues of a Cooperative Cache with No Coherence Problems. In *Proceedings of the 5th Workshop on I/O in Parallel and Distributed Systems* (1997), pp. 37–46.

[10] Forney, B. C., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Storage-Aware Caching: Revisiting Caching for Heterogeneous Storage Systems. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002).

[11] Garg, S. TFLOPS PFS: Architecture and Design of a Highly Efficient Parallel File System. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (1998), pp. 1–12.

[12] Gropp, W., Thakur, R., and Lusk, E. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, 1999.

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

[14] Henning, J. L. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Computer 33*, 7 (2000), 28–35.

[15] James V. Huber, J., Chien, A. A., Elford, C. L., Blumenthal, D. S., and Reed, D. A. PPFS: a high performance portable parallel file system. In *Proceedings of the 9th International Conference on Supercomputing* (1995), pp. 385–394.

[16] Kallahalla, M., and Varman, P. J. Optimal Prefetching and Caching for Parallel I/O Sytems. In *Proceedings of the 13th annual ACM Symposium on Parallel Algorithms and Architectures* (2001), pp. 219–228.

[17] Kandemir, M. A Collective I/O Scheme Based on Compiler Analysis. In *Proceedings of the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers* (2000), pp. 1–15.

[18] Kandemir, M., and Choudhary, A. Compiler-Directed I/O Optimization. In *Proceedings of the 16th International Symposium on Parallel and Distributed Processing* (2002), p. 19.2.

[19] Kotz, D. Disk-directed I/O for an Out-of-core Computation. In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing* (1995), pp. 159–166.

[20] Moore, F. Disk companies pricing themselves out of business-again: lessons of the past still unlearned. *Computer Technology Review* (March 2003).

[21] Mowry, T. C., Demke, A. K., and Krieger, O. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation* (1996), pp. 3–17.

[22] Nieuwejaar, N., and Kotz, D. The Galley Parallel File System. In *Proceedings of the 10th International Conference on Supercomputing* (1996), pp. 374–381.

[23] Paleczny, M., Kennedy, K., and Koelbel, C. Compiler Support for Out-of-Core Arrays on Data Parallel Machines. In *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation* (McLean, VA, 1995), pp. 110–118.

[24] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. Informed Prefetching and Caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (1995), pp. 79–95.

[25] Pinheiro, E., Bianchini, R., and Dubnicki, C. Exploiting Redundancy to Conserve Energy in Storage Systems. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems* (2006), pp. 15–26.

[26] Pugh, W. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (1991), pp. 4–13.

[27] PVFS2 Development Team. Parallel Virtual File System, Version 2. `http://www.pvfs.org/pvfs2-guide.html`, September 2003.

[28] Schmuck, F., and Haskin, R. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st Conference on File and Storage Technologies* (January 2002), pp. 231–244.

[29] Seamons, K. E., Chen, Y., Jones, P., Jozwiak, J., and Winslett, M. Server-Directed Collective I/O in Panda. In *Proceedings of Supercomputing* (1995).

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

[31] Thakur, R., Bordawekar, R., Choudhary, A., Ponnusamy, R., and Singh, T. PASSION Runtime Library for Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference* (1994), pp. 119–128.

[32] Thakur, R., Gropp, W., and Lusk, E. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation* (1999), pp. 182–189.

[33] Thakur, R., Gropp, W., and Lusk, E. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems* (1999), pp. 23–32.

[34] Thakur, R., Gropp, W., and Lusk, E. Optimizing Noncontiguous Accesses in MPI-IO. *Parallel Computing 28*, 1 (2002), 83–105.

[35] Vilayannur, M., Sivasubramaniam, A., Kandemir, M. T., Thakur, R., and Ross, R. B. Discretionary Caching for I/O on Clusters. In *IEEE International Symposium on Cluster Computing and the Grid* (2003), pp. 96–103.

[36] Wolfe, M. J. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[37] Wong, T. M., and Wilkes, J. My Cache or Yours? Making Storage More Exclusive. In *USENIX Annual Technical Conference* (2002), pp. 161–175.

[38] Yadgar, G., Factor, M., and Schuster, A. Karma: Know-it-All Replacement for a Multilevel Cache. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies* (2007), pp. 25–25.

[39] Zhang, Y. Z., Kung, E. H., and Haworth, D. C. A PDF Method for Multidimensional Modeling of HCCI Engine Combustion: Effects of Turbulence/Chemistry Interactions on Ignition Timing and Emissions. In *Proceedings of the 30th International Symposium on Combustion* (2004), pp. 2763–2771.

[40] Zhu, Q., and Zhou, Y. Power-Aware Storage Cache Management. *IEEE Trans. Comput. 54*, 5 (2005), 587–602.