Scientific Programming 21 (2013) 65–78
DOI 10.3233/SPR-130365
IOS Press

Compiler-directed file layout optimization for hierarchical storage systems [1]

Wei Ding [a,*], Yuanrui Zhang [b], Mahmut Kandemir [a] and Seung Woo Son [c]
[a] *The Pennsylvania State University, University Park, PA, USA*
E-mails: {wzd109, kandemir}@cse.psu.edu
[b] *Intel Corporation, University Park, PA, USA*
E-mail: yuanrui.zhang@intel.com
[c] *Northwestern University, Evanston, IL, USA*
E-mail: sson@eecs.northwestern.edu

Abstract. File layout of array data is a critical factor that effects the behavior of storage caches, and has so far taken not much attention in the context of hierarchical storage systems. The main contribution of this paper is a compiler-driven file layout optimization scheme for hierarchical storage caches. This approach, fully automated within an optimizing compiler, analyzes a multi-threaded application code and determines a file layout for each disk-resident array referenced by the code, such that the performance of the target storage cache hierarchy is maximized. We tested our approach using 16 I/O intensive application programs and compared its performance against two previously proposed approaches under different cache space management schemes. Our experimental results show that the proposed approach improves the execution time of these parallel applications by 23.7% on average.

Keywords: File layout, compiler optimization

1. Introduction

Caching file blocks in memory (called “storage caching”) is a promising way of alleviating disk latencies in applications that manipulate disk-resident data sets. Modern high end systems can employ storage caching in multiple layers, e.g., compute nodes, storage nodes, and I/O nodes. Specifically, the compute nodes are used to execute the application threads, the storage nodes are used to connect the disks, and the I/O nodes are used to transfer data between compute nodes and storage nodes. As illustrated in Fig. 1, in this type of storage systems, a cluster of compute nodes is connected to a cluster of I/O nodes; each compute node can have a shared or private cache; and each I/O node can have a cache that could be accessed by multiple compute nodes. And, each storage node can employ a cache that is shared by all compute nodes that have access to it. Clearly, how to take advantage of such storage cache hierarchy for I/O-intensive scientific applications may be critical from the performance point of view.

Compiler support for I/O-intensive applications have been investigated in the past. Prior compiler research in this direction can be roughly divided into two categories: *code transformations* and *file layout optimizations*. Most of the prior code transformation work [4,25,27] deals with iteration space tiling and distribution of loop iterations across multiple nodes. File layout optimizations [27] represent a complementary approach and prior efforts in this direction target sequential applications. To our knowledge, there is no prior compiler-based work that automatically optimizes file layouts targeting parallel computing platforms with hierarchical storage caches. Note that, while it is possible to apply single node centric file layout optimizations to multi-threaded applications that run on parallel systems, such an approach would have at least two problems. First, single-node centric file layout optimization strategies fail to capture data sharing patterns among parallel threads, and consequently, the resulting file layouts may not be very good when access patterns of all compute nodes are considered. Second, as will be demonstrated later in this paper through ex-

[1] This paper received a nomination for the Best Paper Award at the SC2012 conference and is published here with permission from IEEE.

[*] Corresponding author. E-mail: wzd109@cse.psu.edu.

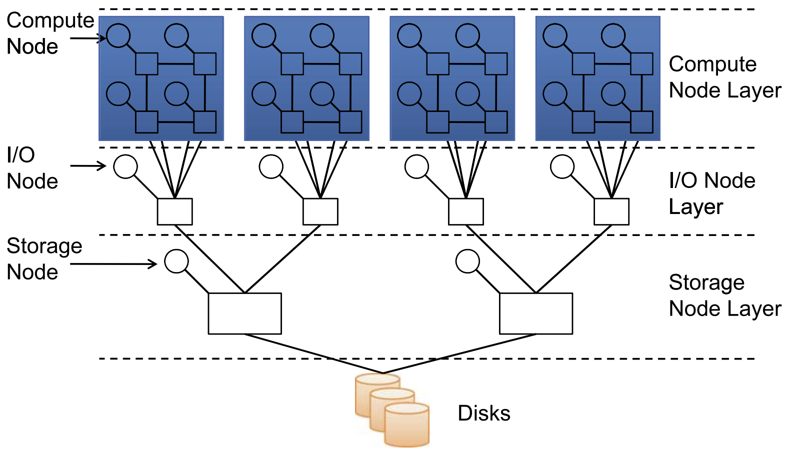

Fig. 1. A sample storage system with three-layer storage cache hierarchy. Rectangles represent caches at different layers. Applications run on CPUs on compute nodes. I/O nodes aggregate I/O demands from compute nodes and issue aggregated I/O requests to storage nodes. The set of storage nodes (or file servers) manage storage devices and provide high performance I/O. In this hierarchy, the I/O nodes run a file system client on behalf of the compute nodes. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130365.)

perimental analysis, optimizing layout for only a single layer versus for an entire storage cache hierarchy (multiple layers) can lead to very different results. Therefore, a framework that optimizes file layouts considering parallel (inter-thread) data access patterns and underlying hierarchical cache organization can be very useful in practice. This paper is a step in this direction and makes the following contributions:

• Targeting high-performance computing platforms and I/O-intensive, array-based parallel applications that manipulate disk-resident data sets, we propose a *fully automated* file layout optimization strategy. This strategy focuses on data layout restructuring as opposed to code restructuring, and it is implemented within an optimizing compiler and determines an optimized file layout for each disk-resident data array manipulated by the application. To the best of our knowledge, this is the first compiler-based I/O work that determines a file layout which considers parallel file accesses from multiple threads running on high performance computing systems with hierarchical storage caches.

• We evaluate this optimization strategy using a set of 16 I/O applications under different system configurations that accommodate hierarchical caches. Our experimental results indicate that the proposed approach improves (in our default system configuration) execution latencies by 23.7% when averaged over 16 I/O-intensive applications. Our results also show that this approach outperforms state-of-the-art code transformation and file layout optimization.

• We demonstrate that the effectiveness of our approach increases when recent exclusive cache management strategies are employed.

2. Multi-layer storage hierarchy

Our approach targets multi-layer storage hierarchy. Figure 1 illustrates an example of such hierarchy with three layers: compute nodes, I/O nodes, and storage nodes. Recent high-end parallel computing platforms, such as the IBM Blue Gene/P (BG/P) system [18] at Argonne National Laboratory the Cray XT5 system [19] at Oak Ridge National Laboratory, employ multi-layered systems. Designing robust high-end systems is becoming increasingly challenging, partly because matching storage system performance to high degree of computation parallelism available is becoming increasingly difficult. As far as storage systems are concerned, there is also a growing performance gap between CPU/memory speed and disk latency. Therefore, high degrees of scalability are hardly achieved without providing a set of dedicated nodes for handing I/O. For example, a key concept in the Blue Gene architecture is the organization of compute and I/O nodes, also called an I/O forwarder [1] in BG/P, into logical groups called *processing sets* or *psets*. A pset consists of one I/O node and a collection of compute nodes. The I/O forwarding mechanism built in this architecture dramatically reduces the number of file system operations/clients that the storage system (parallel file system) sees, thereby improving overall system's scalability. The Cray XT5 I/O subsystem also has similar architecture where storage arrays connected to I/O nodes, called SIO nodes, which reside on a high-speed interconnect [19].

In our study, we assume that nodes in all layers can be equipped with certain amount of storage caches (though in our experiments we allocate caches only in I/O and storage nodes, due to the high demand of compute node memory). Multi-layer caching has previously been investigated in several studies [12,44,47]. In Fig. 1, each rectangle represents storage caches at different layers. In this particular configuration, every four compute nodes are connected to one dedicated I/O node, all of which are then connected to two storage server. We note that the ratio of I/O nodes to compute nodes can vary from system to system, and depending on the specific configuration, each I/O node services I/O request from different number of computer nodes. Since multi-layered architectures are prevalent, it is crucial to explore cache management and optimization policies that operate in a *hierarchy-aware* manner.

We now informally discuss what role file layout optimization can play in improving performance of hierarchical storage caches. From a data access perspective, one can talk about three different "layouts" for data. "Array Layout" represents how a multi-dimensional data structure is viewed by the program. "File layout", on the other hand, is a result of the mapping between array elements (in the multi-dimensional space) and file locations. Finally, "Disk Layout", a result of the mapping between file locations and disks, captures the storage order of data in disks, and is influenced by the underlying file striping strategies adopted. While file systems do *not* enforce any specific file layout, straightforward layouts such as "row-major" and "column-major" are probably the most widely used ones. In a row-major layout for instance, data elements in a row are stored in consecutive file locations, and rows are stored one after another. Although such simple file layouts are probably sufficient for sequential applications with good spatial reuse, they may not be the best option for multi-threaded applications that manipulate disk-resident datasets.

To illustrate this, consider Fig. 2(a) which shows a highly simplified example where a thread (of a multi-threaded application) makes data accesses to a file under a default (e.g., row-major) file layout. It can be observed that these data accesses are scattered all over the file space, which tends to increase the number of "data blocks" (our cache management unit) occupied. Higher number of data blocks required to store the requested data elements in turn increases the pressure on shared storage caches. More specifically, since each

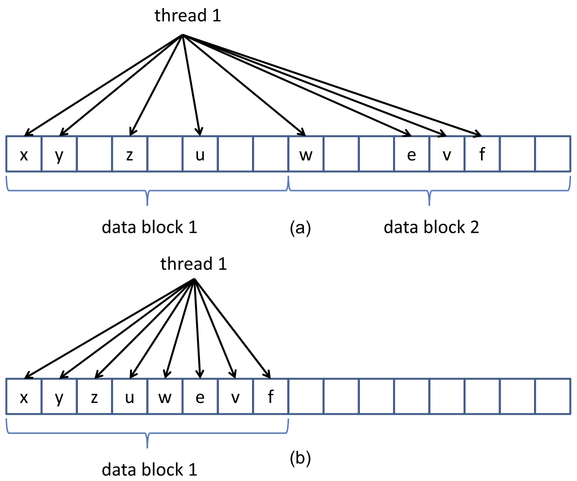

Fig. 2. Importance of optimized file layout in minimizing the number of blocks needed to store the accessed data. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130365.)

thread brings to the cache more data elements than it needs, effective cache capacity is reduced, which in turn affects application performance. Further, this also hurts performance of other threads that share the same storage cache with this thread. Consider now Fig. 2(b) which shows the same data accesses under an alternate file layout optimized using our approach, technical details of which will be presented in the following sections. In this case, the data elements accessed by our thread are stored in consecutive file locations, which helps us minimize the number of data blocks occupied. That is, optimizing file layout in this fashion reduces the "block footprint" of a node in the target hierarchical cache system. Based on this observation, we propose a framework to minimize the number of data blocks accessed by each thread at *each* storage layer in a multi-layer storage hierarchy. It needs to be emphasized however that the final file layout should be determined by considering accesses coming from *all* threads of the application, and in addition, the resulting mapping function between data elements and file locations should be easy to express (from a compiler implementation viewpoint). This optimization is called "inter-node file layout optimization" in this paper, and the file layout determined by it *cannot* be obtained through existing conventional file layout optimization strategies alone.

3. Background

In this section, we give the notations and main concepts used in this paper. Figure 3(a) shows an ex-

```
MPI File open (· · ·,W, &fh W, · · ·);
for i=1, N, 1
   for j=1, N, 1
      W[i, j] +=1;
MPI File write (fh W, · · ·);
MPI File close( &fh W );
```

(a)

```
for i=1, N, 1
   for j=1, N, 1
      W[i, j] +=1;
```

(b)

Fig. 3. (a) An example code fragment with file I/O commands, and (b) its simplified version.

ample code fragment written using MPI-IO [17]. It performs matrix multiplication on disk-resident data. For simplicity, we omit the I/O commands and represent such a code as shown in Fig. 3(b) in our discussion. In such I/O-intensive applications, if the loop bounds and array accesses are affine functions of outer loop indices and loop-invariant parameters (to simplify the notation, we omit the loop-invariant parameters in our discussion), we can employ the *polyhedral model* [2] to represent loops and disk-resident arrays, in which, the iteration space of an n-level loop nest is viewed as an n-dimensional polyhedron bounded by linear inequalities derived from loop bounds. Each point in this polyhedron is denoted by an *iteration vector* $\vec{i} = (i_1, i_2, \ldots, i_n)^T$, with i_k being the iterator of the kth loop (starting from the outermost one), where $1 \leqslant k \leqslant n$. Similarly, the data space of an m-dimensional array is viewed as an m-dimensional polyhedron bounded by constants derived from the array declaration statements, and each point (array element) in this polyhedron is denoted by a *data (index) vector* $\vec{a} = (a_1, a_2, \ldots, a_m)^T$, with a_k corresponding to the index of the kth dimension, where $1 \leqslant k \leqslant m$. Each array reference $\vec{r}$ can be expressed in terms of the iteration vector as: $\vec{a} = Q \cdot \vec{i} + \vec{q}$, where Q is the *access matrix* of $m \times n$ and $\vec{q}$ is the *offset vector* of $m \times 1$. For example, the reference $W[i, j]$ to the disk-resident array W in Fig. 3(b) can be expressed as:

$$\vec{r} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \cdot \vec{i} + \begin{pmatrix} 0 \\ 0 \end{pmatrix},$$

where $\vec{i} = (i_1, i_2)^T$ and $\vec{r} = (r_1, r_2)^T$. In other words, each array reference represents a *mapping* from the iteration space to the data space of the corresponding disk-resident array.

Another important concept used in this work is *hyperplane*. In an x-dimensional space (either iteration or data space) denoted by the vector $\vec{b} = (b_1, b_2, \ldots, b_x)^T$, a hyperplane is defined by an affine equality $g_1 b_1 + g_2 b_2 + \cdots + g_x b_x = c$, where $g_1, g_2, \ldots, g_x$ (rational numbers) are hyperplane coefficients and c is hyperplane constant. The *hyperplane vector* $\vec{g} = (g_1, g_2, \ldots, g_x)$, which is normal to the hyperplane, defines a hyperplane family, where "member" hyperplanes have the same hyperplane vector but different values of c. In this work, hyperplanes are used to express iteration space and data space partitions based on access patterns.

Hyperplanes can be used to explain our loop parallelization and distribution strategy, where, the m-dimensional iteration space is *evenly* partitioned into *iteration blocks* (the last block may have a smaller number of iterations). By a set of parallel hyperplanes orthogonal to the uth dimension (corresponding to the uth loop, where $u \neq m$),[2] and these iteration blocks are assigned to the threads in a round robin fashion in the order of the thread numbers. These parallel hyperplanes are called the *iteration hyperplanes*. The hyperplane vector that represents these iteration space hyperplanes is a $1 \times m$ unit vector (denoted as $\vec{h}_{I,s}$) in the form of $(0, 0, \ldots, 0, 1, 0, \ldots, 0)$, where 1 appears at the uth position. The left portion of Fig. 5 illustrates an example of our loop parallelization and distribution strategy where iteration blocks are distributed across two threads ($P1$ and $P2$).

4. Inter-node file layout optimization

Figure 4 shows where our inter-node file layout optimization falls in the compilation flow. Our optimization is applied following the loop parallelization and distribution phase (described in Section 3), which generates parallelized loop nests that access one or more disk resident arrays. In addition to these loop nests, the input to our approach also includes a description of the target storage cache topology (e.g., the number of layers, number of caches at each layer, their capacities, and block sizes). Each array is assumed to be stored in a separate file.[3] Based on the discussion in Section 2, in order to minimize the number of data blocks accessed by each thread, we need to first find the data elements touched by each thread. This is the main goal in the first step of our approach (Section 4.1). Informally speaking, this step re-orders the data elements stored in the file such that the portion of data elements accessed by each thread can be easy identified based

[2]The value of u is specified by the user.

[3]While storing multiple arrays in the same file can potentially bring further benefits, we postpone its investigation to a future study.

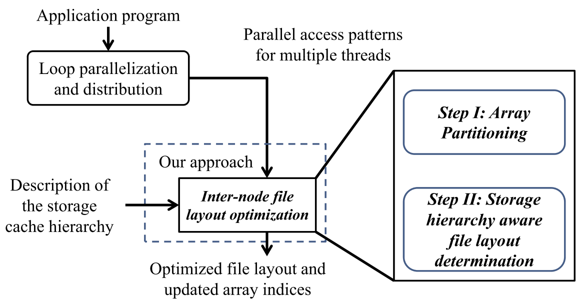
Application program
Loop parallelization and distribution
Parallel access patterns for multiple threads
Our approach
Description of the storage cache hierarchy
Inter-node file layout optimization
Optimized file layout and updated array indices
Step I: Array Partitioning
Step II: Storage hierarchy aware file layout determination

Fig. 4. High level overview of our approach. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130365.)

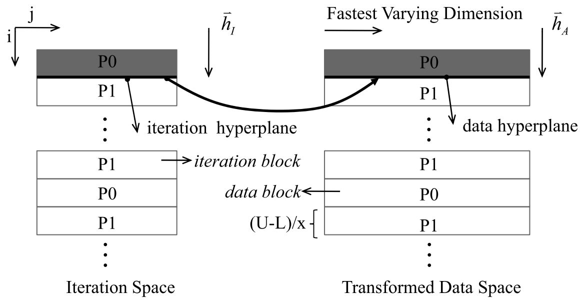

Fig. 5. Illustration of array partitioning.

on certain dimension of the array. After that, our second step (Section 4.2) takes the storage cache hierarchy into account to form the file layout such that the number of data blocks accessed by each thread can be minimized at each storage cache level. The output of our approach are the modified array accesses (array index functions are updated) and optimized file layout for each array based on parallel accesses issued to it by different threads. It is important to emphasize that our approach determines a file layout for each disk-resident data at compile time. There is no dynamic (runtime) layout changes involved and no other runtime overheads.

4.1. *Step I*: *Array partitioning*

Based on our loop parallelization and distribution strategy explained in Section 3, in this section, we perform a unimodular data transformation on each array to isolate the data elements touched by different threads.

Let us first introduce the concept of *unimodular* data transformation. A *unimodular* data transformation maps each data vector $\vec{a}$ in the original data space to a unique vector $\vec{a}'$ in the transformed data space through a transformation matrix D, whose determinant is either $+1$ or -1 [30], i.e., $\vec{a}' = D\vec{a}$, and the array reference $\vec{r}$ is changed accordingly to $\vec{r}\,'$, i.e., $\vec{r}\,' = D\vec{r}$.

The basic idea of array partitioning is to perform a unimodular data transformation on each array to isolate the data elements touched by different threads. Observe that, in our loop parallelization and distribution strategy (explained in Section 3), two iterations (denoted as $\vec{i}_1$ and $\vec{i}_2$, respectively) that reside on the same iteration hyperplane (defined by $\vec{h}_I$) must be accessed by the same thread and grouped into the same iteration block (see Fig. 5). Therefore, if the data elements accessed by these two iterations through a transformed reference $\vec{r}'$ always reside on the same hyperplane with a hyperplane vector $\vec{h}_A$ on the transformed data space, then the transformed n-dimensional data space for each array can be *evenly* partitioned into *data blocks* (the last data block may have smaller number of data elements) by a set of parallel hyperplanes represented by $\vec{h}_A$, such that all the data elements in each data block are only accessed by the same thread. Here $\vec{h}_A$ is a $1 \times n$ unit vector in the form of $(0, 0, \dots, 0, 1, 0, \dots, 0)$, where 1 appears at the vth position, and such a partitioning is illustrated in Fig. 5. Let L and U refer, respectively, to the lower and upper bounds along the vth dimension of the array, and x be the number of iteration blocks in the iteration space. Then, from this figure, we can also see that, the data block size along the vth dimension can be expressed as $(U - L)/x$.

Based on the above observation, let us first derive an expression that D must satisfy for the target array in cases where it has only one (original) reference $\vec{r}$. First, note that, any two iterations $\vec{i}'_1$ and $\vec{i}'_2$ that reside on a hyperplane defined by a hyperplane vector $\vec{h}_I$ (in the transformed iteration space) should satisfy:

$$\vec{h}_I(\vec{i}'_1 - \vec{i}'_2) = 0. \tag{1}$$

Let E_u be the matrix that is obtained from an $m \times m$ identity matrix by deleting the uth row. Since $\vec{h}_I$ us a $1 \times m$ unit vector, i.e., $\vec{h}_I = (0, 0, \dots, 0, 1, 0, \dots, 0)$, where 1 appears at the uth position, each row vector of E_u is a solution for $\vec{i}'_1 - \vec{i}'_2$.

Second, since the two data elements $\vec{a}'_1$ and $\vec{a}'_2$ accessed by these two iterations through $\vec{r}'$ always reside on the same hyperplane defined by $\vec{h}_A$ (in the transformed data space), similar to Eq. (1), we have $\vec{h}_A(\vec{a}'_1 - \vec{a}'_2) = 0$. Assuming $\vec{r} = Q\vec{i} + \vec{o}$ and $\vec{r}' = D\vec{r}$, we further have:

$$\vec{h}_A DQ(\vec{i}'_1 - \vec{i}'_2) = 0. \tag{2}$$

In other words, any two iterations $\vec{i}_1'$ and $\vec{i}_2'$ that satisfy Eq. (1) must also satisfy Eq. (2). As a result, we have:

$$\vec{h}_A DQE_u = 0, \tag{3}$$

Eq. (3) essentially gives the equation that D should satisfy when there only exists one reference for the target array. Since $\vec{h}_A$, Q and E_u are known, Eq. (3) is actually a homogeneous linear system, which can be solved by using Integer Gaussian Elimination [38].

We can further conclude that, in case where there are multiple references to the target array, we have:

$$\begin{aligned} &\vec{h}_A DQ_1 E_u = 0,\\ &\vec{h}_A DQ_2 E_u = 0,\\ &\vdots\\ &\vec{h}_A DQ_k E_u = 0, \end{aligned} \tag{4}$$

where have $Q_1, Q_2, \ldots, Q_k$ are different array access matrices for these references. Equation (4) consists of k homogeneous linear systems to solve. Clearly, a unique D that satisfies all these homogeneous linear systems may not exist. Our strategy is to assign a *weight* to each access matrix to determine the most beneficial linear system to solve first, which in turn, satisfies the majority of references. Specifically, assuming that there are s references in a given set of multiple loop nests that have the same access matrix Q_i, then the weight of Q_i, denoted as $W(Q_i)$, is the *sum* of the number of times that each of these references is accessed, which can be expressed as:

$$W(Q_i) = \sum_{j=1}^{s} n_j, \tag{5}$$

where n_j is estimated by the product of the trip counts (the number of iterations) of the loops that enclose the said reference.

4.2. *Step II: Storage hierarchy aware file layout determination*

At this point, array partitioning has identified the data regions locally accessed by different computation blocks (or threads) on compute nodes. Note however that it does not determine a linear file layout for each array to be stored in disks. Therefore, our second step is to create a linear layout for each array based on array partitioning. An important feature of this step is that it explicitly considers the storage-cache hierarchy in the target I/O architecture and forms a linear file layout that minimizes the number of file blocks used by all threads at any given time as well as conflicts between threads in shared storage caches, thus reducing expensive I/O cache misses. The created (linear) file layout can also help improve the effectiveness of hardware I/O prefetching if supported by the underlying system.

The key point to form this file layout is to construct a *layout pattern* according to the storage cache hierarchy in the target architecture. Specifically, this layout pattern is a thread-interleaved pattern built in a top-down fashion, i.e., from compute nodes to I/O nodes and then to storage nodes that connect to disks, with the consideration of cache capacities at different layers.[4] To make it easy to follow, we use the architecture depicted in Fig. 6(c) and an already-partitioned two-dimensional array for four threads/computation blocks to describe our strategy. In Fig. 6(c), we assume that there are four compute nodes, each running a thread; however, no storage caching is employed in each compute node. Therefore, we omit compute nodes and only depict two layers, where SC1 holds two I/O nodes (each connecting to two compute nodes) and SC2 holds one storage node that connects to disks. Suppose that the storage cache capacities at SC1 and SC2 are S_1 and S_2, respectively, and we have $S_1 < S_2$. We now discuss how to form the layout pattern under these assumptions.

First, we construct a SC1 pattern $\langle P_1, P_2\rangle$ of size S_1. This pattern has two contiguous file chunks, each of which has size $S_1/2$. The first file chunk contains $S_1/2$ data elements accessed by thread P_1, and the sec-

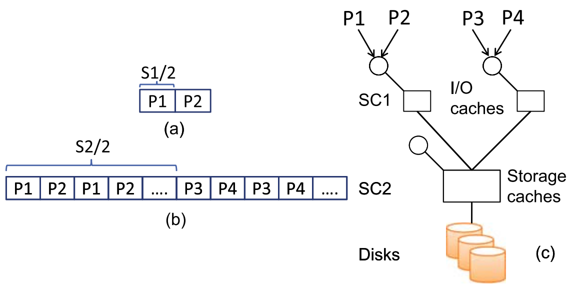

Fig. 6. (a) and (b) are two layout patterns for the sample architecture shown in (c), in which the rectangles denote caches and circles represent processors. Only storage and I/O nodes are shown for clarity. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130365.)

[4]We assume the capacities of different caches that belong to the same layer are the same.

ond file chunk contains $S_1/2$ data elements accessed by thread P_2. Clearly, if we place the data elements accessed by P_1 and P_2 in this way, each of them utilizes half of the shared SC1 cache space for this array access, which helps reduce the conflict misses between P_1 and P_2 at runtime. Similarly, we can also construct a SC1 pattern $\langle P_3, P_4\rangle$ for P_3 and P_4. Next, we construct a SC2 pattern $\langle P_1, P_2, P_3, P_4\rangle$ of size S_2, by first repeating the SC1 patterns $\langle P_1, P_2\rangle$ until size $S_2/2$, and then repeating the SC1 pattern $\langle P_3, P_4\rangle$ for the rest. Clearly, if we place the data elements accessed by P_1, P_2, P_3 and P_4 in this way, the space contention among these threads in the shared SC2 cache can be alleviated. This SC2 pattern is the layout pattern we are looking for, and it will be applied repeatedly to create a linear file layout for the entire array.

After forming these layout patterns, we now discuss how to use these patterns to perform array index transformation. The key point is to find the starting address of each file chunk corresponds to a thread. Let us take thread P_1 in Fig. 6(c) for example. Assuming that its base address is $base_1$, the starting address of its xth file chunk (x starts from 0) can be calculated as $base_1 + b_1 + b_2$, where $base_1$ is the starting address of corresponds to P_1, b_1 and b_2 are the starting addresses of patterns $\langle P_1, P_2, P_3, P_4\rangle$ and $\langle P_1, P_2\rangle$ that contain the xth file chunk, respectively. Let t_1 be the number of times that $\langle P_1, P_2\rangle$ repeats inside $\langle P_1, P_2, P_3, P_4\rangle$, i.e., $t_1 = S_2/2S_1$, then we have $b_1 = (x\%t_1)S_1$ and $b_2 = (x/t_1)S_2$.

The starting address of the xth file chunk of other threads can be calculated in the same way. In general, in a symmetric n-layer cache hierarchy with evenly-distributed threads across compute nodes, let S_i be the storage cache capacity at the ith layer, N_i be the number of caches that a SCi cache connects to at layer $(i-1)$ (e.g., $N_2 = 2$ in Fig. 6(c), since an SC2 cache is connected to two SC1 caches), l be the number of threads per SC1 cache (or compute node), and t_i represent the number of times a SCi pattern repeats inside a SC$(i+1)$ pattern ($t_i = S_{i+1}/(N_{i+1}S_i)$). We have $b_i = ((x/(t_1 \cdots t_{i-1}))\%t_i)S_i$, where $1 \leqslant i \leqslant n-1$, and $b_n = (x/(t_1 \cdots t_{n-1}))S_n$. The starting address of the xth file chunk of P_t can be calculated as: $base_t + b_n + b_{n-1} + \cdots + b_2 + b_1$. The pseudo-code for this storage cache hierarchy-aware file layout formation/indexing is given in Algorithm 1.

4.3. Discussion

We next discuss the limitations of our work. First, in the proposed approach, we treat all out-of-core data as temporary. In other words, the mapping of array data to file stream only exist in the image and is optimized for a particular number of threads and storage hierarchy. Therefore, the data is not readable by other applications. One possible way to solve this problem is to add two more layout transformations to our approach (one for input and one for output arrays). Specifically, the input arrays can be transformed – at the beginning of the program – from a canonical layout (e.g., row major) and the output arrays – at the end of the program – can be transformed either into a canonical layout or into a layout that is desired by the application that will use those arrays as input.

Second, in the proposed approach, recompilation is needed when the system-parameters are changed, such as storage cache hierarchy, cache size, and number of I/O caches. However, we are working on an extension that generates layout for a "template hierarchy" instead of a specific (concrete) hierarchy. For example, all hierarchies with the same number of high-level caches connected to a low-level cache can be considered as belonging to the same "template", and a single compilation for all architectures that belong to the same template would suffice (with some performance loss, of course).

Algorithm 1. Inter-node file layout optimization

```
1:  INPUT: $k$ threads and with $n$ level storage caches.
2:  OUTPUT : File layout for each array.
3:  for each array $A_i$ do
4:      determine its transformation matrix $T_i$ by using Eq. (4).
5:      for $j = 0 \to k-1$ do
6:          $base_j = Addr(A(\ldots, a_{u-1}, (j+1)(U-L)/x, a_{u+1}, \ldots))$;
7:          for each data elements $\vec{a}$ accessed by thread $j$ do
8:              if $counter == S_1/p$ then
9:                  $x_j{+}{+}$;
10:                 $b_1 \leftarrow (x_j\%t_1)S_1$;
11:                 $\vdots$
12:                 $b_{n-1} \leftarrow ((x_j/(t_1 \cdots t_{n-2}))\%t_{n-1})S_{n-1}$;
13:                 $b_n \leftarrow (x_j/(t_1 \cdots t_{n-1}))S_n$;
14:                 $addr_j \leftarrow base_j + b_n + b_{n-1} + \cdots + b_1$;
15:                 $Addr(\vec{a}) \leftarrow addr_j$;
16:                 $counter \leftarrow 0$;
17:             else
18:                 $addr_j{+}{+}$; $counter{+}{+}$;
19:             end if
20:         end for
21:     end for
22: end for
```

5. Experiments

5.1. Setup and applications

We performed our experiments using a Linux cluster that runs MPI-IO [17] on top of the PVFS parallel file system [7]. 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. While as explained earlier, our approach is quite general and determines a file layout for any given storage cache hierarchy, in our experimental evaluation we focus on a three-tier system (of the type shown in Fig. 1), and assume that only I/O and storage layers have caches. The major configuration parameters used in our experimental evaluation and their default values are given in Table 1. Later in our experiments, we change the values of some of these parameters and carry out a sensitivity analysis. We used a set of 16 multi-threaded scientific applications that manipulate disk-resident data. Brief descriptions and important characteristics of these applications are listed in Table 2. These applications are collected from different sources and their common characteristic is that they are I/O-intensive. Apsi, applu, swim and mgrid are out-of-core versions of well-known SPECOMP [39] applications, and similarly, bt and sp are *out-of-core* versions of the corresponding NAS benchmarks [34]. S3asim and qio are frequently used I/O benchmarks [35]. cc-ver-1 and cc-ver-2 are two different implementations for protein structure prediction, and afores is the I/O template for an alternative fuel combustion simulation code (these codes are locally maintained). Twer is a twister simulation kernel, and hf, sar and contour are implementations of the Hartree–Fock method, synthetic aperture radar kernel, and contour display, respectively. These applications use different programming interfaces (e.g., MPI-IO, PnetCDF, HDF5); but, all use PVFS beneath this interface. The cache statistics and execution times presented in the last three columns of Table 2 are for the "original applications" without any file-layout optimization (but caches are used in both I/O node and storage node layers), when using the setup shown in Table 1. Also, in this default execution, each thread of the application is assigned to a single compute node, and we thus have a total of 64 threads (although our approach can also work with multiple threads per compute nodes).

Table 1
Major system parameters and their default values for our target architecture

Parameter	Value
Number of compute nodes	64
Number of I/O nodes	16
Number of storage nodes	4
Data striping	Uses all 4 storage nodes
Stripe size	128 kB
Storage capacity/disk	40 GB
RPM	10,000
Data block size	128 KB
Cache capacity/node (I/O, storage)	(1 GB, 2 GB)

Note: The system architecture is similar to the one shown in Fig. 1 (except for the number of nodes at different layers).

Table 2
Our applications, storage cache misses, and execution times (under the "default execution")

Application name	Miss rate		Execution time
	I/O caches (%)	Storage caches (%) (%)	
cc-ver-1	6.1	4.4	3 min 21 s
s3asim	7.4	6.6	3 min 36 s
twer	29.0	44.9	5 min 27 s
bt	16.2	29.4	1 min 44 s
cc-ver-2	27.9	21.6	4 min 59 s
astro	52.2	61.3	6 min 18 s
wupwise	36.4	52.5	3 min 24 s
contour	31.9	64.2	4 min 07 s
mgrid	13.3	38.4	5 min 31 s
swim	34.8	19.9	2 min 57 s
afores	26.7	24.5	7 min 12 s
sar	22.6	57.9	6 min 14 s
hf	39.1	41.6	5 min 41 s
qio	18.2	26.8	2 min 28 s
applu	44.2	26.1	4 min 05 s
sp	46.4	37.0	8 min 50 s

We implemented our storage caches at I/O and storage nodes. For this purpose, a portion of the main memory in each node is reserved to keep copies of the frequently-used data. The unit of granularity for managing these caches is a data block whose value is the same as the stripe size (at the storage node level). These storage caches are managed using the LRU policy and are inclusive. Note that, while the results we present below depend on the specific caching policy employed (e.g., LRU; inclusive), our approach itself can work with any storage caching policy, that is, it is orthogonal to how the storage caches in the system are managed. Later in this section we also present results with two state-of-the-art exclusive caching policies.

We implemented our file layout optimization algorithm using the SUIF compiler framework [40]. Our optimization increased compilation times of the original applications by about 36% on average, and the highest compilation time observed was about 50 seconds. For each application program in our experimental suite, in addition to the "default execution" which uses the "original file layouts", we performed experiments with our proposed inter-node file layout optimization.

Before we start presenting our experimental evaluation, we want to discuss how our approach improved file layouts of different data arrays manipulated by these multi-threaded benchmark codes. The number of disk-resident arrays in our codes ranges from 3 (in benchmark afores) to 17 (in benchmark twer). When considering all the codes tested, our approach was able to optimize about 72% of these arrays on average. In particular, we were able to optimize the layouts of all arrays in benchmark s3asim.

5.2. *Results with the default values of our experimental parameters*

Our first set of results, given in Fig. 7(a), present the execution times *normalized* with respect to the default execution described above. Based on the results collected, our applications can be divided into three groups. In the first group, which contains cc-ver-1, s3asim and twer, are applications that do not benefit from inter-node file layout optimization. The underly-

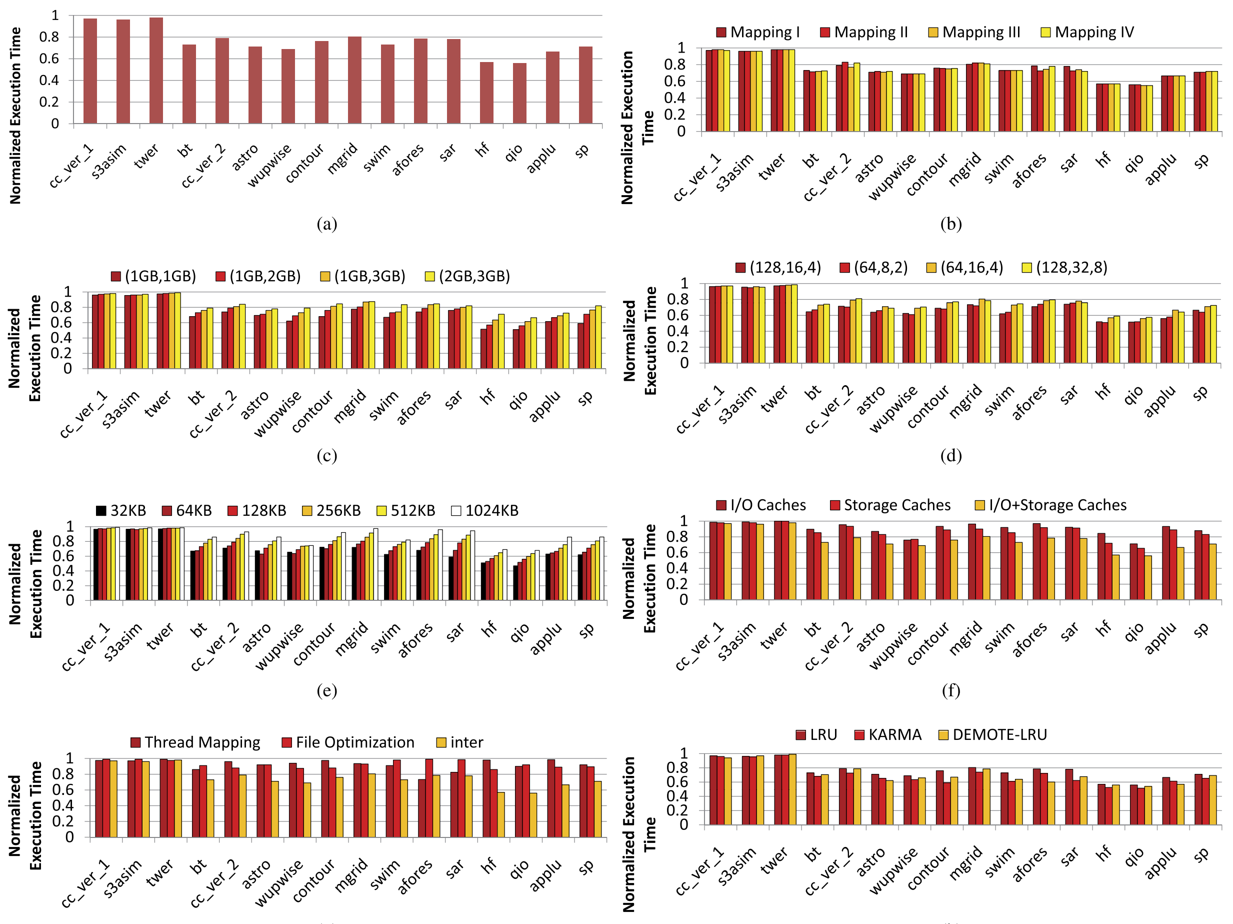

Fig. 7. Experimental results. (a) Execution time results. (b) Results with different thread-to-compute node mappings. (c) Sensitivity to cache capacities. (d) Sensitivity to node counts at different layers. (e) Sensitivity to block size. (f) Sensitivity to the number of layers targeted. (g) Comparison against alternate schemes [26] (first bar) and [27] (second bar). (h) Comparison against prior hierarchical cache management schemes [47] (KARMA) and [44] (DEMOTE-LRU). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130365.)

Table 3

The cache misses after our approach has been applied

Name	I/O caches	Storage caches
cc-ver-1	0.88	0.91
s3asim	0.92	0.94
twer	0.94	0.98
bt	0.52	0.59
cc-ver-2	0.62	0.71
astro	0.54	0.51
wupwise	0.58	0.66
contour	0.63	0.59
mgrid	0.71	0.74
swim	0.59	0.64
afores	0.63	0.76
sar	0.67	0.72
hf	0.48	0.58
qio	0.43	0.61
applu	0.57	0.59
sp	0.63	0.66

Notes: The values are normalized with respect to the corresponding values in Table 2.

ing reason for this can change from one application to another. For example, cc-ver-1 and s3asim already have very good cache hit rates in their default execution; there is simply no scope for additional performance improvement. In comparison, in twer, overly-conflicting requests from different threads at different points in execution prevent the compiler from choosing a good file layout. The applications in the second group, which contains bt, cc-ver-2, astro, wupwise, contour and mgrid, benefit reasonable well from the inter-node layout optimization (with improvements ranging between 8% and 13%). In the third group, which includes applications swim, afores, sar, hf, qio, applu and sp, the inter-node layout optimization brings significant benefits (between 21% and 26%). When all applications are considered our compiler-based file layout optimization scheme brings an average execution time improvement of 23.7%. To explain these results, we present in Table 3 the normalized misses in I/O and shared caches.

5.3. *Sensitivity experiments*

Our goal in this section is to present and discuss results from our sensitivity experiments. In each experiment presented below, only one parameter value is modified; the remaining parameters retain their default values given in Table 1.

In our first set of sensitivity experiments, we try different thread-to-compute node mappings. In the results plotted in Fig. 7(b), Mapping I denotes our default thread mapping used in our experiments so far (and is reproduced here for ease of comparison). On the other hand, Mapping II, Mapping III and Mapping IV represent different (alternate) thread mappings. Different mappings of threads to compute nodes (which are actually different random permutations of threads to compute nodes) are not expected to generate significantly different results from one another in these applications, mostly due to the data parallel nature of computations. In three applications however, namely, cc-ver-2, afores and sar, where the parallel computations mostly implement a master-slave model rather than a data-parallel model, thread mapping makes a difference. Even in these cases however, the difference among different thread mappings remains within 6%, indicating that the behavior of our approach is largely independent of the underlying thread-to-compute node mapping employed. In other words, our approach successfully tailors the file layout to any given parallel data access pattern.

Next, we measure the sensitivity of our schemes to the cache capacities employed at different layers. Recall from Table 1 that, by default, we have 1 GB and 2 GB caches in the I/O and storage layers, respectively. It can be observed from Fig. 7(c) that, when the cache sizes are small, our approach brings more improvements. This is because a smaller cache capacity makes it more critical to exploit data locality. The next parameter we study is the number of nodes at different layers of the storage hierarchy. Recall from Table 1 that we have 64 compute nodes, 16 I/O nodes, and 4 storage nodes in our default configuration. We use $(64, 16, 4)$ to refer to this default configuration. Figure 7(d) plots the execution time improvements under different configurations. Note that individual cache capacities are as shown in Table 1. One observation we can make from these results is that our approach is more successful when there is more pressure on I/O and storage caches, that is, when they are shared by more client and I/O nodes, respectively. This is because the careful management of available cache space becomes more important under high sharing.

We next study the sensitivity of our savings to the data block size. Recall that, the data block is the cache space management unit, and is also the stripe size used in our experiments. The results in Fig. 7(e) indicate that working with smaller block sizes tends to improve the performance benefits brought by our approach. This is expected as a smaller block size allows a finer granular management of available cache space, which in turn leads to better file layouts for the disk-resident datasets.

The graph in Fig. 7(f) plots the normalized execution times when our approach is applied targeting I/O nodes only (the first bar), storage nodes only (the second bar), and both layers (the version evaluated so far; the third bar). One can clearly observe from these results that targeting the entire storage hierarchy is critical. In other words, targeting individual layers in the storage hierarchy is not sufficient. In fact, targeting only I/O node layer and storage node layer result in average performance improvements of 9.1% and 13.0%, respectively, which are much lower than the case where we target both the layers (23.7%).

5.4. *Comparison against prior work*

In this section, we first compare our approach to two previously-proposed compiler-guided data locality optimization strategies. The first of these strategies [26] implements an intelligent loop iteration distribution strategy that targets at exploiting locality in hierarchical storage caches. Specifically, the scheme described in [26] implements an iterative strategy, which clusters loop iterations based on the topology of the underlying storage cache hierarchy and how client nodes share caches that reside at different layers of the hierarchy. Note that this is a *computation restructuring* strategy, as opposed to our approach, which is a data (file) layout optimization strategy. The second strategy [27] proposes a file layout optimization method (in addition to computation restructuring) for I/O intensive applications, but does not target explicitly hierarchical storage systems. This strategy basically implements a profiler-based *dimension reindexing* method, using which, for example, the compiler can convert a row-major file layout to a column-major one. In our implementation of this strategy, using profiling, we exhaustively tried all possible dimension reindexings (e.g., for a three-dimensional disk-resident array, six possible file layouts) and selected the one that generated the best execution time. The normalized execution times (again, with respect to the default execution) achieved with these schemes are presented in Fig. 7(g), along with the results obtained when using our proposed approach (inter). We can make two observations from these results. First, our layout based approach is more successful than the computation mapping. This is primarily because most disk-intensive scientific applications implement data parallel computations where computation mapping is not the most critical factor. The result is that computation mapping achieves about 7.6% improvement (compared to 23.7% obtained using our approach). Our second observation is that the prior file layout optimization [27] leads to an improvement of 7.1% on average. The reason why this value is much below than what is obtained with our approach is because the layouts determined by our approach *cannot* simply be expressed as a dimension reindexing or a combination of several dimension reindexings applied one after another. As explained earlier, our approach determines a file layout by considering parallel data accesses coming from all compute nodes (and the underlying storage cache hierarchy), which is something *cannot* be achieved through dimension reindexing.

So far in our experimental analysis, we tested our approach under an LRU based hierarchical cache management policy with inclusive caches. It is important to note that our strategy determines an optimized file layout considering accesses coming from parallel threads. Therefore, in principle, this strategy can be used along with any cache hierarchy management strategy. To check this, we also implemented two previously proposed hierarchical cache management schemes ([47] and [44]) and tested them under our file layout optimization strategy. KARMA [47] implements *exclusive caching* by using application hints at all layers to classify all cached blocks into disjoint sets and partition the cache according to this classification. DEMOTE-LRU [44] also implements *exclusive caching* where clients do block demotions, and the storage array employs the traditional LRU cache management for both demoted and recently read blocks. In our experiments, we applied these caching schemes only to the I/O and storage layers, which are also the layers targeted by our approach. In the results presented in Fig. 7(h), the second bar for each application indicates the execution time achieved when our approach is used with [47], normalized to the result obtained when using [47] without our approach. Similarly, the third bar represents indicates the execution time achieved when our approach is used with [44], normalized to the result obtained when using [44] without our approach. As can be seen from these results, the effectiveness of our layout optimization strategy increases when using these alternate storage hierarchy management schemes. In fact, the average improvements in execution cycles are 30.1% with [47] and 28.6% with [44]. The reason why our approach increases the effectiveness of KARMA is because more localized data accesses enables KARMA to generate more accurate hints. On the other hand, our approach improves the effectiveness of DEMOTE-LRU (as compared to LRU) because it leads to better demotions.

6. Related work

We now discuss previous work related to our file layout optimization.

6.1. *Multi-level caching*

Most prior work [10,37,41,45] focuses on improving behavior of storage (or second-level) cache management because the behavior of the second-level cache is often hard to characterize, making cache management schemes inadequate. Particularly, Zhou et al. [50] show that LRU is not suitable for managing storage cache. To address LRU's poor performance, several techniques, such as multi-queue replacement [49], eviction-based placement policies [8] and CLOCK-pro [20] have been proposed. Choi et al. propose a fine-grained file-level characterization of chunk references in buffer management [10]. Vilayannur et al. present selective caching because caching of certain block is not always beneficial [41]. Sarhan and Das propose to use the on-disk buffers for caching intervals between successive streams, while multimedia-on-demand servers improves resource sharing by intelligent request schedulers [37]. Our approach complements any existing caching policies with improved cache locality because of file layout transformation.

Recently, many studies [3,9,16,21,28,44,46] looked into cache management for multi-level storage hierarchies. The main motivation for these studies is that the modern networked storage systems have a hierarchy of caches, and special care needs to be taken in order to manage those cache hierarchies efficiently. A key idea is how to reduce negative interference while keeping most valuable blocks in shared cache [46]. Techniques to extract and predict the most valuable blocks include transforming application-level requirement into I/O reservations [3], correlating program counters with program context [16], exploiting reference regularities [28], locality of file chunks of non-uniform strength [23], and automatic application reference pattern detection [9]. Wong and Wilkes [44] explore the exclusive cache policies against the prevalent inclusive ones. These studies are system-level approaches and are, therefore, orthogonal to our approach.

6.2. *Exploiting data access pattern*

Another set of research efforts aim to use access pattern history or other hints for I/O performance improvement [6,14,22,24,33,36,42,47,48]. Proposed techniques infer or predict future access patterns using a guest OS's buffer cache information [24], chunk access history [48], hints from an application (or client) [6,33,36,47], and temporal and spatial locality in buffer cache [14,22]. The inferred access pattern can then be used for dynamic and/or adaptive cache management policies. Our approach would increase effectiveness of access pattern extraction mainly because transformed file layouts could help improve cache locality.

6.3. *I/O prefetching*

Prefetching and its effects on shared storage caches have also been extensively studied [5,11,13,15,29,31,32]. These studies showed inefficiencies of existing replacement algorithms [5,11], and proposed several techniques such as managing prefetch memory for online servers [31], Diskseen [11], self-tuning adaptive cache management policy [15], combined prefetching and caching for systems with multiple disks [29]. More recent studies proposed studied the problems in sequential prefetching and proposed adaptive asynchronous algorithms [13] and TaP [32]. All these studies investigate how to prevent prefetched pages from being evicted before being used. These studies are similar to ours because the main goal is to improve storage cache performance, but they are not explicitly target multi-level storage cache hierarchies. Unlike these studies, our approach is based on compiler analysis and file layout changes.

6.4. *Compiler support for out-of-core computations*

The compiler techniques have been extensively explored in I/O and studied blocking/tiling [43], out-of-core compilation that efficiently block data movement between storage and memory [4,25], etc. Kandemir et al. [27] propose a compiler-based approach to optimize cache behavior in the I/O server only. Chang and Gibson used speculative execution technique as an extension of TIP [36]. The work most relevant to our study was conducted by Kandemir et al. [26], who used a compiler to map computation for multi-level storage cache hierarchies. Our approach is similar to this computation mapping in that we also use a compiler analysis, but our approach is to optimize file layouts rather than restructuring computations. technique is more focused on file layout optimization. To the best of our knowledge, this is the first work that uses a compiler to optimize file layout to optimize the performance of multi-level storage cache hierarchies.

7. Concluding remarks

The main contribution of this paper is a compiler-driven file layout optimization strategy that targets hierarchical storage caches. By analyzing a given application code, the proposed strategy determines parallel accesses to shared disk-resident arrays and changes the mapping between the array elements and linear file space such that the data elements accessed by a thread are stored in consecutive file locations minimizing the number of data blocks needed in the shared cache space. We tested the effectiveness of the proposed compiler algorithm using 16 I/O intensive applications and different hierarchical storage configurations. The collected experimental data are very promising and indicate that our approach generates significant performance improvements under various multi-layer cache management policies. Our results also show that the proposed approach outperforms prior code and file layout optimization based strategies.

Acknowledgements

This work is supported in part by NSF grants CNS 1017882, CNS 1152479, CCF 0937949, CCF 0833126 and DOE grant DE-SC0002156.

References

[1] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward and P. Sadayappan, Scalable I/O forwarding framework for high-performance computing systems, in: *Proceedings of the IEEE International Conference on Cluster Computing*, 2009, pp. 1–10.

[2] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, O. Temam, A. Group and I. Rocquencourt, Putting polyhedral loop transformations to work, in: *Workshop on Languages and Compilers for Parallel Computing*, 2003, pp. 209–225.

[3] D.O. Bigelow, S. Iyer, T. Kaldewey, R.C. Pineiro, A. Povzner, S.A. Brandt, R.A. Golding, T.M. Wong and C. Maltzahn, End-to-end performance management for scalable distributed storage, in: *Proceedings of the 2nd International Workshop on Petascale Data Storage*, 2007, pp. 30–34.

[4] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel and M. Paleczny, A model and bordawekar-ooc for out-of-core data parallel programs, in: *Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995, pp. 1–10.

[5] A.R. Butt, C. Gniady and Y.C. Hu, The performance impact of kernel prefetching on buffer cache replacement algorithms, in: *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, 2005, pp. 157–168.

[6] P. Cao, E.W. Felten and K. Li, Implementation and performance of application-controlled file caching, in: *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 1994.

[7] P.H. Carns, W.B. Ligon, III, R.B. Ross and R. Thakur, PVFS: A parallel file system for Linux clusters, in: *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000, pp. 391–430.

[8] Z. Chen, Y. Zhou and K. Li, Eviction based cache placement for storage caches, in: *Proceedings of the USENIX Annual Technical Conference*, 2003.

[9] J. Choi, S.H. Noh, S.L. Min and Y. Cho, An implementation study of a detection-based adaptive block replacement scheme, in: *Proceedings of the USENIX Annual Technical Conference*, 1999, pp. 18–18.

[10] J. Choi, S.H. Noh, S.L. Min and Y. Cho, Towards application/file-level characterization of block references: a case for fine-grained buffer management, *SIGMETRICS Perform. Eval. Rev.* **28** (2000), 286–295.

[11] X. Ding, S. Jiang, F. Chen, K. Davis and X. Zhang, DiskSeen: exploiting disk layout and access history to enhance I/O prefetch, in: *Proceedings of the USENIX Annual Technical Conference*, 2007, pp. 20:1–20:14.

[12] B.S. Gill, On multi-level exclusive caching: offline optimality and why promotions are better than demotions, in: *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008.

[13] B.S. Gill, L. Angel and D. Bathen, AMP: Adaptive multi-stream prefetching in a shared cache, in: *Proceedings of the Fifth USENIX Symposium on File and Storage Technologies*, 2007, pp. 185–198.

[14] B.S. Gill, M. Ko, B. Debnath and W. Belluomini, STOW: a spatially and temporally optimized write caching algorithm, in: *Proceedings of the USENIX Annual Technical Conference*, 2009, p. 26.

[15] B.S. Gill and D.S. Modha, SARC: sequential prefetching in adaptive replacement cache, in: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005, pp. 33–33.

[16] C. Gniady, A.R. Butt and Y.C. Hu, Program-counter-based pattern classification in buffer caching, in: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, 2004, pp. 27–27.

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

[18] IBM Blue Gene/P, available at: http://www.ibm.com/watson.

[19] Jaguar, available at: http://www.nccs.gov/jaguar/.

[20] S. Jiang, F. Chen and X. Zhang, CLOCK-Pro: an effective improvement of the clock replacement, in: *Proceedings of the USENIX Annual Technical Conference*, 2005, p. 35.

[21] S. Jiang, K. Davis and X. Zhang, Coordinated multilevel buffer cache management with consistent access locality quantification, *IEEE Transactions on Computers* **56**(1) (2007), 95–108.

[22] S. Jiang, X. Ding, F. Chen, E. Tan and X. Zhang, Dulo: an effective buffer cache management scheme to exploit both temporal and spatial locality, in: *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies*, 2005.

[23] S. Jiang and X. Zhang, Ulc: a file block placement and replacement protocol to effectively exploit hierarchical locality

in multi-level buffer caches, in: *Proceedings of the 24th International Conference on Distributed Computing Systems*, 2004, pp. 168–177.

[24] S.T. Jones, A.C. Arpaci-Dusseau and R.H. Arpaci-Dusseau, Geiger: monitoring the buffer cache in a virtual machine environment, in: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 14–24.

[25] M. Kandemir, A. Choudhary, J. Ramanujam and R. Bordawekar, Compilation techniques for out-of-core parallel computations, *Parallel Comput.* **24** (1998), 597–628.

[26] M. Kandemir, S.P. Muralidhara, M. Karakoy and S.W. Son, Computation mapping for multi-level storage cache hierarchies, in: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 179–190.

[27] M. Kandemir, S.W. Son and M. Karakoy, Improving I/O performance of applications through compiler-directed code restructuring, in: *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008, pp. 11:1–11:16.

[28] J.M. Kim, J. Choi, J. Kim, S.H. Noh, S.L. Min, Y. Cho and C.S. Kim, A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references, in: *Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation*, 2000.

[29] T. Kimbrel, A. Tomkins, R.H. Patterson, B. Bershad, P. Cao, E.W. Felten, G.A. Gibson, A.R. Karlin and K. Li, A trace-driven comparison of algorithms for parallel prefetching and caching, in: *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, 1996, pp. 19–34.

[30] S. Leung and J. Zahorjan, Optimizing data locality by array restructuring, Technical report, Department of Computer Science and Eng., University of Washington, 1995.

[31] C. Li and K. Shen, Managing prefetch memory for data-intensive online servers, in: *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies*, 2005.

[32] M. Li, E. Varki, S. Bhatia and A. Merchant, TaP: table-based prefetching for storage caches, in: *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008.

[33] X. Li, A. Aboulnaga, K. Salem, A. Sachedina and S. Gao, Second-tier cache management using write hints, in: *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies*, 2005.

[34] NAS Parallel Benchmarks, available at: http://www.nas.nasa.gov/Resources/Software/npb.html.

[35] Parallel I/O Benchmarks, available at: http://www.mcs.anl.gov/˜thakur/pio-benchmarks.html.

[36] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky and J. Zelenka, Informed prefetching and caching, in: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995, pp. 79–95.

[37] N. Sarhan and C. Das, An integrated resource sharing policy for multimedia storage servers based on network-attached disks, in: *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003, pp. 136–143.

[38] A. Schrijver, *Theory of Linear and Integer Programming*, Wiley, 1998.

[39] SPEComp, available at: http://www.spec.org/omp/.

[40] The SUIF 2 Compiler System, available at: http://suif.stanford.edu/suif/suif2/.

[41] M. Vilayannur, A. Sivasubramaniam, M. Kandemir, R. Thakur and R. Ross, Discretionary caching for I/O on clusters, in: *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2003, pp. 96–103.

[42] M. Wachs, M. Abd-El-Malek, E. Thereska and G.R. Ganger, ARGON: performance insulation for shared storage servers, in: *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, 2007.

[43] M.J. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley/Longman, 1995.

[44] T.M. Wong and J. Wilkes, My cache or yours? Making storage more exclusive, in: *Proceedings of the General Track of the USENIX Annual Technical Conference*, 2002, pp. 161–175.

[45] C.H. Xia, D. Towsley and C. Zhang, Distributed resource management and admission control of stream processing systems with max utility, in: *Proceedings of the 27th International Conference on Distributed Computing Systems*, 2007.

[46] G. Yadgar, M. Factor, K. Li and A. Schuster, MC2: Multiple clients on a multilevel cache, in: *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, 2008, pp. 722–730.

[47] G. Yadgar, M. Factor and A. Schuster, KARMA: know-it-all replacement for a multilevel cache, in: *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, 2007.

[48] J. Yoon, S.L. Min and Y. Cho, Buffer cache management: Predicting the future from the past, in: *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks*, 2002.

[49] Y. Zhou, Z. Chen and K. Li, Second-level buffer cache management, *IEEE Trans. Parallel Distrib. Syst.* **15** (2004), 505–519.

[50] Y. Zhou, J. Philbin and K. Li, The multi-queue replacement algorithm for second level buffer caches, in: *Proceedings of the USENIX Annual Technical Conference*, 2001, pp. 91–104.