Iteration Aware Prefetching for Large Multidimensional Scientific Datasets

Philip J. Rhodes
Dept. of Computer Science
University of Mississippi
rhodes@cs.olemiss.edu

Xuan Tang
EMC Corporation
tang_xuan@emc.com

R. Daniel Bergeron
Dept. of Computer Science
University of New Hampshire
rdb@cs.unh.edu

Ted M. Sparr
Dept. of Computer Science
University of New Hampshire
tms@cs.unh.edu

Abstract

Most caching and prefetching research does not take advantage of prior knowledge of access patterns, or does not adequately address the storage issues associated with multidimensional scientific data. Armed with an access pattern specified at run time as an iteration over a multidimensional array stored as a disk file, we use prefetching to greatly reduce the number of disk accesses and mitigate the cost of read latency. We call this iteration aware prefetching.

We assume the pattern of access is not known until runtime, in contrast to chunking methods that preprocess a file for a particular access pattern. Our approach results in dramatic performance improvements over file system caching. We also significantly outperform chunking without having to reorganize the data, and can do even better by applying our approach on top of a chunked file.

1 Introduction

Scientists often work with data represented in an *n*-dimensional space in which data values are associated with a location in the space [Cigno97, Hibbard95]. For example, satellite data is typically considered to be organized in a two dimensional space, while medical CT and MRI data usually exists in a three dimensional space. We consider these kinds of scientific data to be *multidimensional*. Multidimensional data presents special challenges when designing efficient access methods because elements that are nearby in the data space may not be nearby in the underlying data file. The caching and prefetching schemes present in most operating systems do not take into account the natural spatial relationships in the data, so they tend to cache, discard, or prefetch the wrong information.

Over the last fifteen years there has been a thousandfold increase in processor speed, along with even larger gains in memory and disk capacity. During the same period, the size of scientific data sets increased even into the terabyte range. However, the average seek time of hard disk drives has improved only modestly over the same period [Coughlin, Chang01]. The work described here is motivated by the need to minimize the now comparatively high latency or *stalling* costs associated with modern disk drive media. Using our system, a researcher can take advantage of improved I/O performance without spending time on the minutiae of efficient file access.

To implement this abstraction while still maintaining efficiency, the researcher must be able to define the application's data access pattern. We are developing a toolkit of *iterators* that succinctly describe the access pattern and also perform the iteration through the data space. Using knowledge of the access pattern, we can create a cache and a prefetching strategy that usually provides significant speedup for the application.

A unique aspect of our approach is that we create and prefetch cache blocks with *n*-dimensional shape, as opposed to the 1 dimensional pages of file system caches and similar methods. *N*-dimensional cache blocks can be given a shape that is tuned to a particular iteration and to the storage organization of the data. We choose a shape that minimizes the total number of disk accesses while reading data that is sure to be visited in the near future by the iteration. We call this method *spatial prefetching*, an example of *iteration aware prefetching*.

Unlike other methods for achieving efficient I/O performance [Sarawagi94, More00], our approach does not require any reorganization of the data. That is, we work with the original data file, rather than making a copy with a different storage organization.

The work described here is done in the context of the *datasource* component of the Granite Scientific Database System, which is in turn an implementation of our multisource multiresolution data model for scientific data [Rhodes01]. The datasource layer handles multidimensional data in which sample points are arranged in a regular and rectilinear fashion throughout the domain. As with many other scientific databases, the design of the Granite system assumes that *update* operations are infrequent or entirely absent, so the work described here is aimed toward a read-only data environment.

After a brief overview of related work, the next several sections describe the functionality and implementation of the datasource, iterator and cache classes, all of which contribute to the support of transparent and efficient out-

This work is supported by the National Science Foundation under grants IIS-0082577 and IIS-9871859

of-core access. We then present performance test results that demonstrate the significant advantages of this approach. Finally, we end with future work and conclusions.

2 Related Research

Providing efficient access to huge scientific datasets is a challenging problem, and has attracted a lot of attention from both the operating system and scientific data management communities. Work has focused on either providing comprehensive scientific data management systems, or optimizing file systems using techniques like prefetching, caching and parallel I/O.

2.1 File Access

Reorganizing datasets on disk to speed access has been explored by a number of researchers. Sarawagi and Stonebraker [Sarawagi94] describe *chunking*, which uses the expected access pattern to group spatially adjacent data elements into *n*-dimensional chunks which are then used as a basic I/O unit, making access to multidimensional data an order of magnitude faster. They also arrange the storage order of these chunks to minimize seek distance during access. Following this work, many other reorganization methods have been developed. More and Choudary [More00] reorganize their data according to the expected query type, and the likelihood that data values will be accessed together. The Active Data Repository (ADR) uses chunking to reduce overall access costs and to achieve balanced parallel I/O [CChang00, CChangADR].

2.2 Prefetching and Caching

Software prefetching has been used by many researchers to hide or minimize the cost of I/O stalling. In the file systems arena, approaches to this problem can be distinguished by whether or not prefetching is guided by explicit information about the access pattern. Albers *et al*. [Albers98] describe an algorithm that produces an optimal schedule for prefetching and discarding cache blocks when the entire access pattern is given in advance. Other researchers have explored the case where the access pattern is disclosed less completely in the form of *hints*. Patterson *et al*. [Patterson95] developed a framework for informed caching and prefetching based on a cost-benefit model. This model has been extended to account for storage devices with very different performance characteristics [Forney02]. Cao *et al*. demonstrate success by letting applications have control of data cache replacement strategy in their share of cache blocks [Cao96]. Brown *et al*. [Brown01] describe a hint based method that effectively accelerates paged virtual memory performance using an operating system that takes advantage of compiler generated hints and multiple disks. Kotz [Kotz97] describes *disk directed I/O*, a method for aggregating and prefetching data requests in a parallel environment. Mowry [Mowry94] presents software controlled prefetching for hiding or reducing the latency experienced by a processor accessing memory.

When no explicit information about access pattern is available, the history of prior accesses can be used to predict future accesses. Amer *et al*. group files together based on historical file access patterns [Amer02]. Other researchers have used probability trees or graphs to represent the likelihood of future block accesses given past and current block accesses [Vellanki99, Highley03, Griffioen94]. Madhyastha *et al*. use a hidden Markov model to automatically predict file access patterns over time; the file system adaptively selects appropriate caching and prefetching policies according to the detected pattern [Madhyastha96, Madhyastha97].

At the application level, Chang [Chang01] adds a separate thread to the user program that performs prefetching by mimicking the I/O behavior of the main thread and preloading data. Doshi [Doshi03] describes a system that adaptively selects a prefetching strategy based on user behavior. The VisTools [Nadeau] system is most similar to our approach. It provides an application level data prefetching and caching service for huge multidimensional datasets, using the Paged-Array schema. It reads formatted pages of elements from the underlying files when the first element in the page is requested. The formatted pages are then stored in a *page cache* for fast future re-access. When the cache size limit is reached, the paged-arrays are deleted or written to a swap file. Like our own work, paged-arrays also support intelligent prefetching guided by the iterators that have an *n*-dimensional view of the dataset. However, the one dimensional nature of pages fails to take into account the proximity of elements in *n*-dimensional space. By using pages as its unit of cache storage, VisTools and other page based methods may make poor decisions about what data to retain or discard. The following section examines this issue in greater detail.

2.3 Advantages of the Granite Approach

Reorganizing data into chunks is a very effective and general technique, but the required reorganization (and implied duplication) of the dataset can be inconvenient, especially when working with large datasets. Also, performance may suffer if the data is accessed in a different way than was expected when the reorganization was performed. The approach adopted by the Granite system works with the original data, and requires no such reorganization.

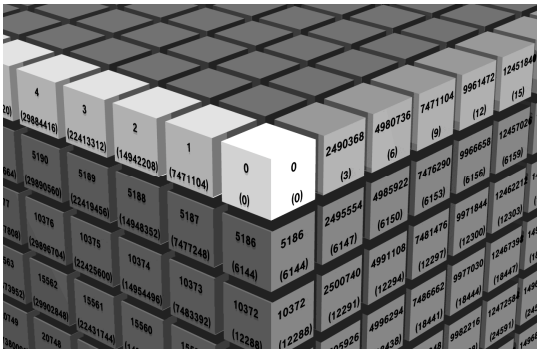

Figure 1. The elements fetched by the file system (medium gray, on the right) are not the elements needed by the iteration (light gray, on the left). The top number in each block indicates iteration order, while the numbers in parentheses indicate the offset in the file.

Systems that access the data in pages suffer from not taking into account the multidimensional nature of the data. In particular, elements that are nearby in *n*-dimensional space may be far apart in the one dimensional file space. Since paging is essentially a one dimensional method, it may be inefficient for an *n*-dimensional access pattern.

Figure 1 shows a conceptual view of a portion of the 39GB Visible Woman dataset, provided by the National Institutes of Health. This dataset consists of 5186x2048x1216 elements of 3 bytes apiece [Rhodes05]. Each block in the figure represents a single element. The number in parentheses at the bottom of each block represents the byte offset of that element from the beginning of the file and the number at the top of each block represents the order in which that block will be visited by an iterator. The light gray blocks show the initial path of the iteration, beginning with the white corner element. When this initial element is accessed, the file system will load a page of data, typically 4K in size, indicated by the series of medium gray elements. Unfortunately, the next element to be visited (the light gray block at offset 7471104) is not contained in this page, so the disk must be read again. In fact, separate reads must be made for each element in the light gray series, since they are all separated by over 7MB in the one dimensional file. When, at step 5186, the iteration returns from the far end of the data set to begin a new "row", a new page will still have to be read, since this element is not contained in any of the pages that have been read so far. In fact, the next medium gray element would be used at step 2490368 of the iteration, but this first page will certainly have been discarded at this point, unless the file system is able to retain 10GB of disk pages in memory. This is clearly beyond the capacity of today's commonly available systems.

File systems also commonly prefetch pages following an explicitly accessed page in the hope that the prefetched pages will be accessed next, and reads to disk will be reduced. This helps slightly in this example, because elements that are vertically adjacent are only 6K apart. So, if the file system prefetches at least one additional page for each read to disk, the elements immediately beneath the light gray elements will be read from pages loaded during the traversal of the light gray elements. Although this is an improvement, it still means that only two elements will be read out of each 4K page before it is discarded, only to be reread later in the iteration. For datasets with even larger dimensions, the distance between vertically adjacent elements may be too large for pages to ever be used more than once before being discarded. In this case, prefetching just makes the situation worse by increasing the number of inappropriate pages loaded into memory. Because the file system has no information about the dimensionality of the data or the path of the iteration, it is grossly unsuited for the job.

We address these issues by creating cache blocks that are *n*-dimensional, and shaped according to the iteration. Elements that are contiguous in the file are loaded in a single *read()* call. This method has several beneficial effects. First, it uses more data from each file system page that is read, thereby reducing the number of redundant reads made to disk. Second, it reduces the number of *read()* calls made to the operating system. Third, since the cache block can be filled in any order, we choose to fill it in a way that most closely matches the ordering of the data in the file. This allows us to sometimes take advantage of the file system prefetching that is otherwise a liability.

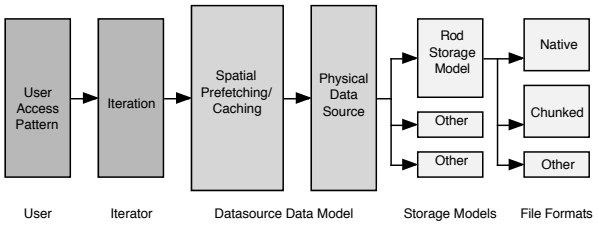

Figure 2. Spatial prefetching and the Datasource data model serves as a bridge between the user access pattern expressed as an iteration, and the data as it lies on disk.

We utilize nearly complete information about the access pattern given by our iterators. We don't have to guess which data to prefetch, and we don't discard needed data before it is used. Because of this, the various caches we have developed require at most two cache blocks to be maintained in memory at a time, which can extend the reach of an application to much larger datasets than would otherwise be possible.

3 The Multidimensional Data Model

Figure 2 is a conceptual diagram of the pipeline relating a user access pattern to the file. The datasource is the representation seen by a Granite user, and uses a *storage model* to help translate the *n*-dimensional data space to the one dimensional file space. The storage model may be able to work with more than one *file format*. For example, the rod storage model discussed later in this section represents both chunked data files and files that have been left in their native plane-row-column order.

3.1 Datasources

We model the data to be processed as a *datasource*, which is conceptually an *n*-dimensional array containing a set of sample points. We call the space defined by the array indices an *index space*. Each location in the index space has a collection of associated data values, which we call a *datum*. Although our datasource model allows datasources to be built on top of other datasources or to be associated with a network stream, we limit our discussion in this paper to datasources that are associated directly with a file on disk.

Datasources must handle two basic kinds of queries. A *datum query* specifies a single location in the index space, and is satisfied by the return of a single datum. A *subblock query* specifies an *n*-dimensional rectangular region of the index space, and is satisfied by the return of a *data block*, which is conceptually an array of datums, with a dimensionality matching the datasource.

3.2 The Rod Storage Model

While the file is a one dimensional entity, a datasource has an index space that is *n*-dimensional. The datasource is responsible for satisfying queries expressed in its index space by reading data from the file. It must therefore map its index space to file offsets. It does this with the help of an *axis ordering,* which is simply a ranking of axes from *outermost* to *innermost*. "Innermost" and "outermost" suggest position in a set of nested *for* loops used to access the file in its storage order on disk. The innermost axis changes most frequently and is called the *rod* axis when referring to the *storage ordering* of a datasource. Each axis is labeled with an integer that identifies the position of coordinates of that axis in a tuple used to specify locations in the index space. Consequently, an axis ordering is just a list of integers that defines an ordering of axis coordinates from least to most frequently varying. For example, an axis ordering of {0,2,1} indicates that coordinates of axis 0 change least frequently, followed by axis 2, and then by axis 1, which changes most frequently.

The number of separate read requests made to the storage device strongly impacts I/O performance, so it is important to minimize the number of reads when satisfying a subblock query. Toward this end, the rod storage model views the datasource as being conceptually composed of *rods*. A rod is a one dimensional sequence of elements that are contiguous in both the *n*-dimensional index space and the 1-dimensional file space. Consequently, rods are always aligned with the rod axis. Because it is contiguous in the file space, a rod can be accessed in one read. Note that rods are composed of datums when the native file format is used, or whole chunks of datums with the chunked file format. When a subblock query is processed, the requested region of index space is decomposed into a collection of rod segments contained in the region. We then retrieve the subblock data from disk in rod-by-rod fashion where each rod segment corresponds to a separate read. To maximize locality, we read this set of rods according to the storage ordering. In the case where a set of rods is itself contiguous (or nearly so) in the file, we issue only one read and retrieve the entire set of rods in one disk operation.

Like the *order line* model described in [Wu03], the rod storage model does not take into account the physical layout of the file on the disk, but only the logical layout presented by the file system. However we have found that this approximation serves as an effective foundation for our iteration aware prefetching, which shows significant performance improvements over other techniques. In addition, applying the rod model to chunked data and other formats extends the reach of iteration aware prefetching to a wider range of data representations.

4 Iterators

Since our system aims to improve I/O performance for particular access patterns, we use iterators to represent access patterns as well as to perform the actual iteration through the datasource index space. Iterators have a value that changes with each call to the iterator's *next()* method. This value might denote a single location in the index space, or perhaps a rectilinear region. In either case, the iterator value can be used directly in datum and subblock queries.

The pattern of iteration is determined when the iterator is constructed. An axis ordering is used to help represent the behavior of iterators that proceed through the index space in rectilinear fashion. In this context, the innermost axis of the iteration is called the *run axis*. While the datasource is conceptually composed of rods, the space being traversed by a rectilinear iterator is conceptually composed of *runs*.

The *iteration space* is the space traversed by the iterator. It may be the entire index space of a datasource, or

some subset of that space. We also represent the starting point and the *stride* through the iteration space in cases where the iterator skips over some locations. The iteration space, starting point, stride and axis ordering all contribute to the creation of a prefetching cache that is tuned to the iteration.

5 Iteration Aware Prefetching

As noted in section 2, much of the literature in caching and prefetching concerns when to load new blocks from disk, and choosing blocks to be discarded. Because we have nearly complete information about the access pattern from the iterator, these problems are vastly simplified in our system. We call our approach *Iteration Aware Prefetching*.

The standard caching and prefetching view of files as one dimensional entities is not adequate for scientific applications involving multidimensional datasets because it misses the neighborhood relationships inherent in the data. The problem becomes even more acute as the dimensionality of the dataset increases. To address this issue we have designed a *multidimensional cache* that preserves the iterator's spatial data view. The iteration space is conceptually partitioned into an *n*-dimensional array of *n*-dimensional cache blocks. Data is read from disk one block at a time, and is retained in memory to quickly satisfy user data requests.

Our iteration aware prefetching approach includes two independent components — *spatial prefetching* and *threaded prefetching*. Their different roles become clear when considering an iterator reading a series of blocks from a datasource. Spatial prefetching reduces the latency costs incurred while reading data from a single block. Threaded prefetching reduces or eliminates the amount of time an application must wait for a complete block to be read by overlapping application processing with I/O.

5.1 Spatial Prefetching

The key contribution of our prefetching strategy is based on adjusting the shape of the cache blocks to minimize the number of separate reads made to disk.

An important characteristic of our approach is that we can perform effective prefetching using a single conceptual cache block. Although there are sometimes practical reasons for breaking a single conceptual cache block into 2 or more physical blocks, the discussion in this section addresses the construction of a single conceptual block that is tuned to the user's access pattern.

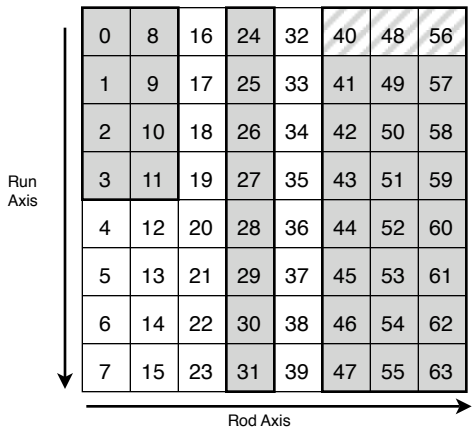

Figure 3. For a {1,0} iteration over a {0,1} datasource, the shape on the right is the only one that is both well formed and practical.

5.1.1 Examples

Figure 3 shows three potential cache block shapes. The numbered sequence indicates a column-by-column iteration over a datasource stored in row-by-row fashion.

Suppose that the shaded shape in the upper left region of the figure were assigned to the cache block. Such an assignment would be poorly suited for a single block cache since step 4 of the iteration causes the block to be discarded, only to be reloaded at step 8. The algorithm shown in figure 4 would extend the block shape all the way down to the bottom of the space before attempting to extend it in the horizontal direction. The cache block shape shown is poorly suited to a single block cache because step 4 of the iteration causes the block to be discarded, only to be reloaded at step 8.

The middle shaded shape in figure 3 does not have this problem, since it extends over the full length of the vertical axis. However, this block cannot reduce the number of read operations. Since the rod axis is the horizontal axis, filling this cache block would require eight separate reads, which is the same number needed with no cache at all.

The shaded shape on the right is much better, since it can be filled with 8 reads of length 3. The striped region represents a single rod subset for this block. Depending on the characteristics of the platform, this shape may produce a useful increase in performance.

5.1.2 Well Formed Cache Blocks

Typically, when a cache needs to load data from disk to satisfy a request, it loads a larger set of data in the neighborhood of the original request. Hopefully, the nearby data can be used to satisfy future requests without returning to the disk. If the pattern of future accesses is already known, however, we can choose a cache block shape that guarantees that all the needed contents will be used before being

Algorithm A1:
Input:
Iterator Ordering $A_n = \{a_0, a_1, a_2, \ldots a_{n-1}\}$,
Iteration space dimensions $S_n = \{s_0, s_1, s_2, \ldots s_{n-1}\}$,
available memory M

Output:
A set of cache block dimensions $B = \{b_0, b_1, b_2, \ldots b_{n-1}\}$ that represent a cache block shape that is well formed with respect to the iterator ordering.

Note:
M(B) indicates the bytes occupied by a cache block of shape *B*

begin
$B = \{1, 1, 1, \ldots 1\}$

for $i = n\text{-}1$ **downto** 0 // *from innermost to outermost*

$axis = a_i$ // *for the next innermost axis…*
$b_{axis} = s_{axis}$ // *extend B to end of iteration space*

if $(M(B) > M)$ **then** // *is memory exceeded?*
$b_{axis} = 1$ // *then return B to previous shape*
$b_{axis} = M / M(B)$ // *extend B as far as memory allows*

done
end
end
end

Figure 4. Algorithm A1 produces a cache block shape that is well formed with respect to a given iteration.

discarded. We say such a cache block is *well formed* with respect to the iteration. A more formal definition follows:

Definition D1:
We denote a rectilinear iteration *I* over a rectilinear iteration space *D* using an axis ordering *A* as *I(A,D)*. Consider a rectilinear region *R* of shape *B* that is a subset of *D*. We say the shape *B* is *well formed* with respect to *I(A,D)* if for any region *R* of shape *B* in *D*, once *I* leaves *R*, it will not revisit *R*.

If we can construct a cache containing blocks that are well formed with respect to a given iterator, we can be assured that no cache block will need to be read more than once, and that once the iterator is done with a cache block, we can discard it. Therefore, most iterations only require a single cache block to be used at one time. Overlapping block iterators require at least two cache blocks, as does threaded prefetching.

Algorithm A1 generates a well formed cache block shape for a datum iterator that visits single elements in the index space. It must be given the iterator ordering, the space over which the iterator travels, and the amount of memory that is available for constructing a cache block.

The algorithm works by marching through the iterator's axis ordering from innermost to outermost axis, setting the corresponding dimension of the cache block shape to equal the extent of the iteration region along that axis. Below is a proof that algorithm A1 produces a well formed cache block shape for a datum iterator.

Proof P1:
Claim: Algorithm A1 produces a well formed shape *B* for the given iterator, iteration space, and available memory.

Base Case: An n-dimensional shape with a single element, $B_0 = \{1, 1, 1 \ldots 1\}$ is well-formed with respect to an iterator $I(A, D)$ using any axis ordering *A*.

Assumption: Algorithm A1 produces a shape at step *k*, B_k, that is well formed with respect to *A*.

Induction step: At step $k+1$, we know that block B_k extends across the entire extent of the iteration space for the *k* least significant axes and that it is well-formed. Algorithm A1 then extends the block along axis $k+1$ to either the entire extent of the iteration space in that axis or as much as will fit in the available memory. In both cases the shape is well formed since the iteration will not return to that shape after leaving it. If the algorithm cannot add the entire extent of the iteration space in that axis, it terminates, leaving a well formed block.

The algorithm and proof can be easily modified to account for block iterators rather than datum iterators. Since block iterators represent a sequence of block accesses, we can set the initial dimensions of the cache block shape to match a single iterator block. The algorithm then proceeds as before. The proof still holds for this case if we consider an element to be a block instead of a single position in the index space. The block version of the algorithm can also be used to handle the case where an iterator has gaps or overlap between visited elements.

5.1.3 Practicality

Whether the shape of a cache block is well formed is related only to a particular iteration. It is possible that a well formed cache block will not enhance performance with a certain dataset because of the way the data lies on disk. In order to guard against this possibility, we must check to see if a cache block shape is *practical* with respect to the storage model. We currently only consider the rod storage model, and our definition of practicality concerns the extent of the cache block shape along the rod axis.

Definition D2:
A cache block shape is *practical* with respect to a rod storage model if it has extent greater than *r* elements along the rod axis, where the value of *r* is determined by cache overhead and the performance characteristics of the I/O subsystem, and must be greater than 1.

This definition is motivated by the fact that in order to get any gain in performance, we must reduce the number of reads made to disk. It follows that we must therefore make each read longer than would be performed without the cache. The extent of the cache block shape along the rod axis determines the length of these reads, so this value must be sufficiently long to provide a performance gain, even in the face of cache overhead.

5.2 File Formats

When the rod storage model is used on top of the native file format, the rods consist of a series of datums stored sequentially on disk. We refer to this file format as "native" because it requires no preprocessing — the file is handled "as is". In this situation, using a well formed cache block also guarantees that no data is read from disk more than once. This is because the cache block is defined in terms of the same units (datums) as the file format.

The rod storage model can also be used on top of chunked files. In this case, the rods consist of a series of contiguous chunks that can be loaded with a single read operation. Here, the file format is defined in terms of units different from what was used to define the cache block. Because of this, data may be read more than once, even with well formed cache block shapes. Currently, we solve this problem by ensuring that for each dimension a cache block will either extend through the entire iteration space, or have length equal to one chunk. This ensures that the cache block is well formed with respect to the *n*-dimensional chunked space.

5.3 Threaded Prefetching

Threaded prefetching uses a separate I/O thread to fetch the next cache block while the current one is being processed. Unlike other systems using I/O threads, we don't have to guess which block should be read next, because that information is contained in the iterator. Currently, we have only implemented and tested threaded prefetching for a single disk, so we can achieve at most the doubling of performance that occurs when the I/O time perfectly matches the computation time for each block.

Our current approach is very effective in hiding the cost of loading a block of data from disk, but even greater performance improvements should be possible if multiple disks are available.

When the rate at which an application consumes data is less than or equal to the rate at which data can be read from disk, threaded prefetching can yield performance similar to the in-core case. The combination of threaded prefetching (even with only one I/O stream) and spatial prefetching can be particularly effective in an interactive application. Figure 5 shows an image created with the

Figure 5. A view of the 39GB Visible Woman Dataset produced with *Slicer*, an interactive slice based volume visualizer.

Slicer application described in [Rhodes05]. *Slicer* uses a combination of threaded and spatial prefetching to view progressive slices of a user defined subset of the 39GB Visible Woman dataset. By matching the frame rate to the capabilities of the I/O subsystem, threaded prefetching allows for smooth animation.

```
// Create datasource
Datasource ds = Datasource.createDS("8gig.xml");

// Create ordering for iterator
AxisOrdering
       iterOrdering= new AxisOrdering(new int[]{0, 1, 2});

// Create an iterator that traverses the entire datasource
ISIterator
    iter=new ISIterator(ds.getBounds(), iterOrdering);

// Create a spatial prefetching cache for the
// given datasource and iterator
CacheDataSource
    cds = CacheMaker.createCDS(ds, iter, freeMem);

// Create a datum to receive data values.
Datum d = new Datum(ds.getNumAttributes());

// Traverse the entire datasource index space,
// accessing the data through the cache.
for( iter.init(); iter.valid(); iter.next() )
{
    cds.datum(d,iter); // Process datum
}
```

Figure 6. Example code for a datum iteration over a cache.

6 Example Code

Figure 6 shows a small example of a datum iteration using the Granite system. We first create the datasource from an xml file that describes such properties as dimen-

	Datum Iteration over native file				Datum Iteration over chunked file			64^3 Block Iteration over native file			64^3 Block Iteration over chunked file		
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**
Ordering	File System Cache	128MB SP Cache	512MB SP Cache	Max Speed up	512MB LRU Cache	512MB SP Cache	Speed up	File System Cache	512MB SP Cache	Speed up	512MB LRU Cache	512MB SP Cache	Speed up
{0, 1, 2}	9963	868	961	11.5	4628	3634	1.27	705	304	2.3	3288	342	9.6
{1, 2, 0}	16786	1311	1294	13.0	9175	3880	2.4	664	279	2.4	3081	342	9.0
{2, 1, 0}	360000 (est)	11349	3719	96.8 (est)	9607	4485	2.14	7847	2777	2.8	3736	966	3.7

Table 1. Results for a complete traversal of an 8GB file of 1024x1024x2048 floats. Native files are in plane-row-column order, while chunked files consist of 4K chunks. All execution times are in seconds.

sionality, size along each axis, and the number of attributes at each location in the index space. Next, we define an axis ordering and iterator that will traverse the datasource. We are now able to create a cache that is tuned to the iteration we wish to use. Finally, we create a datum object for retrieving data values and perform the iteration.

This code is very flexible, and requires very minimal changes in order to work with different datasources and iterator orderings. To make the code work on another file of entirely different size and shape, we only need to change the name of the *xml* file given in the first line of code. The iteration order is just as easily changed, and an appropriate cache will be created without further thought from the programmer.

This flexibility is especially attractive in situations where a user wants to process a large file using several different traversals. With spatial prefetching, it is a simple matter to create caches that are tuned to each iteration. With preprocessing methods, some compromise must be made when deciding the chunked format, unless the user is willing to make a separate file for each iteration.

7 Results

We have run our tests on a variety of machines and found that machines with fast I/O show smaller performance improvement simply because the I/O is a smaller portion of the total execution time.

We present results from the machine with the fastest I/O available to us. This is a single processor Pentium 4 machine with a 2.4Ghz CPU and 2GB of RAM running the Linux operating system. The disk on this machine is a fast 15,000 RPM SCSI disk with a 3.6ms average read seek time. Though we show here very substantial gains in performance, we saw even greater gains on other platforms, since a fast disk actually minimizes the benefits of spatial prefetching.

The Linux file system cache loads and stores 4k blocks of data from disk whenever a file is accessed. Since the file system cache is persistent across task execution, it is possible for a task to request an I/O block for the first time, but still get a cache hit if another task had previously read that block. Although this is generally a good thing, it is problematic for our testing environment. We therefore ran all tests with a cold (i.e., empty) file system cache. In addition to guaranteeing a consistent environment by always starting with an empty cache, this approach more realistically portrays the behavior that a researcher might expect when dealing with very large datasets.

In the following sections, we present results for both datum and block iteration over the entirety of a three dimensional 8GB dataset. ([Rhodes05] examines subset iteration in an interactive context.) On our test machine, running the unix *cp* command with this dataset takes approximately 400 seconds. The dataset has dimensions 1024x1024x2048, where each datum is a single floating point value. Tests were run on both native and chunked file formats. In all cases, the files had a storage ordering of {0,1,2}.

Table 1 shows our results for three different iterator orderings over both native and chunked file formats. Both datum and block access were tested. We have performed extensive testing with a wide range of machine characteristics, file sizes, and cache sizes. For clarity and simplicity, we present results here for a single 8GB data set on one machine configuration and we concentrate on a cache size of 512MB. Considering the current affordability of memory and the recent introduction of commodity 64 bit machines, we feel 512MB is a reasonable memory cost for working with very large data sets. However we still see significant performance improvements for smaller cache sizes.

7.1 Datum Iteration over Native Files

Our datum iteration tests ran code very similar to the example in section 6. Columns *a* through *d* of table 1 show the execution times for traversals using the file sys-

tem cache and spatial prefetching (SP) caches of 128MB and 512MB. In all three iterator orderings, the SP cache provides a very substantial improvement in performance. Notice that the {0,1,2} ordering shows somewhat less improvement than the other orderings. This is because the file system is prefetching blocks in the same order that the iterator will request them. File system prefetching is much less effective for the other orderings, so our spatial prefetching offers more improvement in these cases. In fact, the file system cache test for {2,1,0} ordering did not complete within twelve hours. We determined that the test was making forward progress in a linear fashion, but very slowly, due to the awkward nature of this access pattern. A very simple C program that mimicked the access pattern for this test but performed no type conversion or copying of data took over 37 hours to run, so we are confident that disk access is causing the excessive runtime. Using a simple extrapolation, we estimated the completion time for the Java implementation using the file system cache to be about 100 hours, and we report this estimated value in the table.

The test with a 512MB Spatial Prefetching cache does considerably better in the {2,1,0} direction than the 128MB cache. For this ordering, the rods span the shortest dimension of the cache block, so increasing the available memory increases the length of the rods, meaning more data is read with each read operation.

Clearly, it would be beneficial to develop an automatic means of choosing how much memory to allocate to a cache based on storage ordering, iterator ordering, system characteristics, and total memory available. We plan to extend the notion of practicality to support this functionality in future work.

7.2 Datum Iteration over Chunked Files

Chunking is a common method for speeding access to spatial data, so it is important to compare spatial prefetching alone with the performance of chunked file access. The chunked format typically divides the file into chunks equal to the file system page size. The dimensions of the chunks are chosen to best suit a particular access pattern [Sarawagi94].

An important assumption of our work is that the user access pattern is not known until runtime. A generic chunking method chooses chunk dimensions that are equal or nearly equal in all directions. This method provides a substantial performance improvement for most access patterns without being tailored specifically to a particular one. We therefore chose to compare spatial prefetching with this form of chunking.

Chunking generally requires some kind of cache in order to be effective with datum access, so we implemented a simple LRU cache that holds a collection of chunks. We compared the performance of our spatial prefetching cache against the performance of this LRU cache. In all of these tests, the memory used for both caches was always 512MB, and the file was in chunked format.

Columns *e* through *g* show the execution times for both caches. Comparing LRU performance with the file system datum iteration in column *a*, it is clear that chunking is a very effective technique. However, we get even better performance by applying spatial prefetching on top of chunked files, especially in the last two orderings listed in the table. On machines with larger disk latency, speedup is substantial even in the first case.

Of even greater interest is the fact that the performance of spatial prefetching over a native file presented in column *c* is markedly superior to the performance of the LRU cache over a chunked file shown in column *e*. For each ordering, spatial prefetching produces speedups of 4.8, 7.1, and 2.6 compared with chunking. That such performance can be achieved without preprocessing or duplicating the file makes spatial prefetching a particularly attractive technique.

7.3 Block Iteration over Native Files

Block iteration involves loading successive *n*-dimensional subsets of the data from disk. The rod storage model by itself facilitates this form of access since it reads rods according to the storage ordering, which improves locality. However, spatial prefetching is still able to provide a useful performance increase by reading data for many blocks at one time. Columns *h* through *j* show the execution times for a 64^3 block traversal over the same dataset used in the previous section.

7.4 Block Iteration over Chunked Files

Our fourth group of tests compared the performance of our spatial prefetching cache over a chunked file with the LRU cache on the same file. Columns *k* through *m* show that spatial prefetching over chunked files provides much more meaningful speedup for block access than for datum access. Since datum access involves many more cache lookup operations, it is likely that in this case, cache overhead erodes gains in I/O efficiency.

8 Conclusions and Future Work

Mismatch between iteration and storage patterns is a well-known problem addressed by many systems in an *ad hoc* manner. Generally, these approaches are based on a one-dimensional view of the data and do not provide a convenient application level interface to the prefetching facility. We have developed a comprehensive environment

for seamless integration of the data access pattern and the prefetching mechanism. The future multidimensional access pattern is specified implicitly during construction of an iterator. The iterator, in turn, is used to determine an effective prefetching strategy tuned for the particular combination of file storage order and iteration pattern.

Spatial prefetching can provide a very meaningful performance increase when large data files are accessed in a rectilinear manner. We have shown that performance is superior to generic chunked file access, yet does not require a preprocessing step. Since spatial prefetching can be used "on the fly", it is particularly well suited to situations where the pattern of access is not known until runtime, or when several different patterns will be used on the same file.

The Granite system lets the user take advantage of the efficiencies of spatial prefetching and other iterator aware prefetching methods while abstracting away the details of storage organization.

For future work, we plan to expand our use of iterators to include traversal through a collection of data values of interest to the experimenter. We will also expand our support of the current iterators to include a way to automatically determine an amount of cache memory that provides a good tradeoff between performance and memory use. Lastly, we are exploring the application of our methods to a distributed context. Since network latency can be even more severe than disk latency, we expect promising results in that environment.

9 References

[Albers98] S. Albers, N. Garg and S. Leonardi, Minimizing Stall Time in SIngle and Parallel Disk Systems, *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pp. 454-462, 1998

[Amer02] A. Amer, D. Long, and R. Burns. Group-Based Management of Distributed File Caches. In *Proc. of the 17th International Conference on Distributed Computing Systems,* 2002

[Brown01] A.D. Brown, T.C. Mowry, Compiler-Based I/O Prefetching for Out-of-Core Applications, *ACM Trans. on Computer Systems, Vol. 19, No. 2,* May 2001

[Cao96] P. Cao and E. Felten, Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling, *ACM Transactions on Computer Systems, vol. 14, No. 4*, 1996

[Chang01] F. Chang, Using Speculative Execution to Automatically Hide I/O Latency, *Ph. D. Dissertation*, Carnegie Mellon University, 2001

[CChang00] C. Chang, T. Kurc, A. Sussman, J. Saltz, Optimizing Retrieval and Processing of Multi-dimensional Scientific Datasets, In *Proc. of the Third Merged IPPS/SPDP Symposiums*. IEEE Computer Society Press, May 2000

[CChangADR] C. Chang, T. Kurc, A. Sussman, J. Saltz, *Active Data Repository Software User Manual*, http://www.cs.umd.edu/projects/hpsl/ResearchAreas/ADR-dist/ADR.htm

[Cigno97] P. Cignoni, C. Montani, E. Puppo, Roberto Scopigno, Multiresolution Representation and Visualization of Volume Data, *IEEE Transactions on Visualization and Computer Graphics, Volume 3, No. 4,* IEEE, Los Alamitos, CA, 1997

[Coughlin] T. Coughlin, High Density Hard Disk Drive Trends in the USA, tech report at http://www.tomcoughlin.com /techpapers.htm

[Doshi03] P.R. Doshi, G.E. Rosario, E.A. Rundensteiner, and M.O. Ward, A Strategy Selection Framework for Adaptive Prefetching in Data Visualization, *Proc. of SSDBM,* 2003

[Forney02] B. C. Forney, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Storage-Aware Caching: Revisiting Caching for Heterogeneous Storage Systems, In *The First USENIX Conference on File and Storage Technologies (FAST '02)*, 2002.

[Griffioen94] J. Griffioen and R. Appleton, Reducing File System Latency Using A Predictive Approach, *University of Kentucky Technical Report #CS247-94*

[Hibbard95] W. L. Hibbard, D. T. Kao, and A. Wierse, Database Issues for Data Visualization: Scientific Data Modeling, *Database Issues for Data Visualization, Proceedings of the IEEE Visualization '95 Workshop, LNCS 1183,* Springer, Berlin, 1995

[Highley03] T. Highley and P. Reynolds, Marginal Cost-Benefit Analysis for Predictive File Prefetching, *Proc. of the 41st Anuual ACM Southeast Conference (ACMSE 2003)*

[Kotz97] D. Kotz, Disk-Directed I/O for MIMD Multiprocessors, *ACM Trans. on Computer Systems,* Vol. 15, No. 1, 1997

[Madhyastha96] T. M. Madhyastha, C. L. Elford, and D. A, Reed, Optimizing Input/Output Using Adaptive File System Policies, In *Fifth NASA Goddard Conference on Mass Storage Systems and Technologies*, September 1996

[Madhyastha97] T. M. Madhyastha, and D. A. Reed, Input/Output Access Pattern Classification Using Hidden Markov Models, In *Workshop on Input/Output in Parallel and Distr. Systems*, Nov. 1997

[More00] S. More, A. Choudhary, Tertiary Storage Organization for Large Multidimensional Datasets, *8th NASA Goddard Conference on Mass Storage Systems and Technologies,* 2000

[Mowry94] T. C. Mowry, Tolerating Latency Through Software-Controlled Data Prefetching, *Ph.D. Dissertation*, Stanford University, 1994

[Nadeau] D. R. Nadeau, An Architecture for Large Multi-Dimensional Data Management, *Scalable Visualization Tools White Paper*, http://vistools.npaci.edu/

[Patterson95] R.H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, Informed Prefetching and Caching. In *Proceedings of the 15*th ACM Symposium on Operating Systems Principles, December 1995, PP. 79-95.

[Rhodes01] P.J. Rhodes, R.D. Bergeron, and T.M. Sparr, A Data Model for Distributed Multisource Scientific Data, *Hierarchical and Geometrical Methods in Scientific Visualization*, Springer-Verlag, Heidelberg, 2001

[Rhodes05] P.J. Rhodes, X. Tang, R.D. Bergeron, and T.M. Sparr, "Out of core visualization using Iterator Aware Multidimensional Prefetching", *Conference on Visualization and Data Analysis 2005*

[Sarawagi94] S. Sarawagi, M. Stonebraker, Efficient Organizations of Large Multidimensional Arrays, *Proc. of the Tenth International Conference on Data Engineering*, Feb. 1994

[Vellanki99] V. Vellanki and A. Chervenak, A Cost-Benefit Scheme for High Performance Predictive Prefetching, *Proc. of Supercomputing '99*, Nov. 1999

[Wu03] K. Wu, W. Koegler, J. Chen, A. Shoshani, Using Bitmap Index for Interactive Exploration of Large Datasets, *Proc. of SSDBM,* 2003