Out of core visualization using iterator aware multidimensional prefetching

Philip J. Rhodes[a] , Xuan Tang[b], R. Daniel Bergeron[c], and Ted M. Sparr[c]

[a]Department of Computer Science, University of Mississippi, Oxford, MS;

[b]EMC Corporation, Hopkinton, MA ;

[c]Department of Computer Science,University of New Hampshire, Durham, NH

ABSTRACT

Visualization of multidimensional data presents special challenges for the design of efficient out-of-core data access. Elements that are nearby in the visualization may not be nearby in the underlying data file, which can severely tax the operating system's disk cache. The Granite Scientific Database System can address these problems because it is aware of the organization of the data on disk, and it knows the visualization method's pattern of access. The access pattern is expressed using a toolkit of iterators that both describe the access pattern and perform the iteration itself. Because our system has knowledge of both the data organization and the access pattern, we are able to provide significant performance improvements while hiding the details of out-of-core access from the visualization programmer.

This paper presents a brief description of our disk access system placing special emphasis on the benefits offered to a visualization application. We describe a simple demonstration application that shows dramatic performance improvements when used with the 39GB Visible Woman Dataset.

CR Categories: E.5 [Files]—I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics Data Structures and Data Types I.3.m[Computer Graphics]: Miscellaneous—Scientific Visualization

Keywords: out-of-core visualization, caching, prefetching, rendering large volume datasets

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[1,2]. For example, satellite data is typically considered to be organized in a two dimensional space, while medical data usually exists in a three dimensional space. 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 in most operating systems do not take into account the natural spatial relationships in the data, so they offer poor support for visualizing scientific data.

In the past, the rendering pipeline was often a significant bottleneck for visualization algorithms. More recently, the availability of powerful and inexpensive graphics hardware has accelerated rendering to the point that data can often be rendered more quickly than it can be read from disk. However, the average seek time of hard disk drives has improved only modestly over the same period[3,4]. In many cases, data I/O has replaced rendering as the new bottleneck for visualization of very large datasets.

The Granite Scientific Database System presents two approaches to the problem of reducing I/O costs: multiresolution data representation and iterator-aware prefetching. Multiresolution data representation allows the

Contact information: (Send correspondence to Philip J. Rhodes)
Philip J. Rhodes: rhodes@cs.olemiss.edu
Xuan Tang: xtang@cs.unh.edu
R. Daniel Bergeron: rdb@cs.unh.edu
Ted M. Sparr: tms@cs.unh.edu

experimenter to examine a coarse overview of the data, and then zoom in to progressively smaller subsets at finer resolutions[1,5]. Iteration aware prefetching, the focus of this paper, provides more efficient access to data stored on disk at any resolution. Both techniques can extend the experimenter's reach to larger datasets than would otherwise be feasible, and can be used in concert to provide an even greater advantage.

One of our primary goals is to hide the details of data access from the visualization programmer, while still providing efficient access to the underlying storage device. To implement this abstraction while still maintaining efficiency, the visualization programmer must be able to define the application's data access pattern. We are developing a toolkit of *iterators* that describe the access pattern and also perform the iteration through the data space. This description of the access pattern can then be used to generate a cache that provides a useful speedup to the application. The cache will be tuned to the particular iteration the visualization requires and to the storage organization of the data, but is also transparent to the application programmer. Freed from the details of data access, the visualization researcher is better able to focus on the visualization technique.

The work described here is done in the context of the *datasource* component of the Granite system, which is in turn an implementation of our multisource multiresolution data model for scientific data[6]. We begin by describing *Slicer*, our application for the visualization of data from the NIH Visible Human project, and the problems it presents for I/O performance. 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 support transparent and efficient out-of-core access for the visualization researcher. We then present results for *Slicer* working on data from the 39 GB Visible Woman dataset. Finally, we end with future work and conclusions.

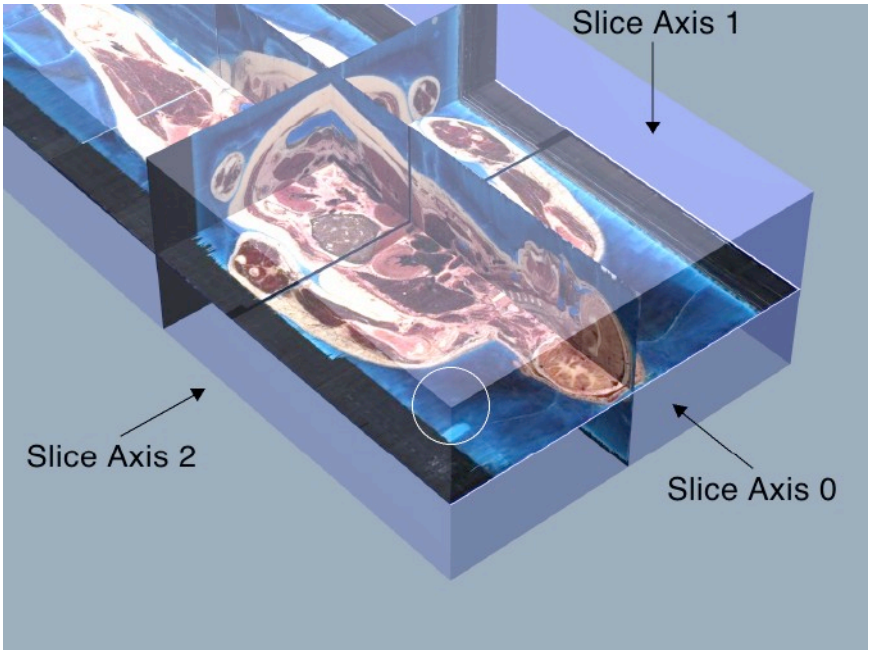


Figure 1. The *Slicer* application can view the Visible Woman dataset from the three principal directions by setting the *slice axis* equal to axis 0, 1, or 2.

2. THE SLICER APPLICATION

We use a simple visualization application to demonstrate the effectiveness of our out-of-core data access system. Our application, called *Slicer,* presents the user with an animated display showing progressive two dimensional *slice planes* of a three dimensional volume. The *slice axis* is orthogonal to the slice plane, and defines the direction of progression through the dataset. Figure 1 shows the three possible slice axes, which must be aligned with the principal axes. The user is able to select the slice axis and the subvolume to be visualized, similar in spirit to the *volume roaming*

described in ref. 7. The 39 GB Visible Woman dataset from the National Institute of Health was used in all tests described in this paper. This dataset has dimensions 5186 x 1216 x 2048 with RGB byte values for each location, giving a total size of 39GB.

When the user chooses to view the volume through slice axis 0, the filesystem cache performs quite well, since this view produces accesses that are contiguous in the one dimensional file space. The filesystem performs less well with slice axis 1, and is almost violently unsuited for the access pattern resulting from a slice axis 2 view.

Figure 2 shows a closeup of the circled corner in Figure 1. The numbers in the figure indicate the one dimensional file offset of the labeled element. The white region is the set of elements contained in the first slice plane for slice axis 2. If we load only the elements in this slice plane, each element requires a separate read, since none of them are neighbors in the one dimensional file space, as can be seen by examining the offsets. In fact, even the elements that are closest to each other are about 6K apart, which is larger than the 4K page size typical on many systems. This means that if we render a 1024x1024 slice plane along slice axis 2, we must load 1024 x 1024 pages of 4K each, for a total of 4GB. Since very few commodity systems have this much memory available, none of the pages loaded for the first slice plane will be resident when the second slice plane is rendered. Those reads will have to be repeated, which leads to a severe degradation in performance.

Filesystems also 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. In this example, this just makes the situation worse, since Slicer is not proceeding through the file space in the way the filesystem expects. Prefetching just increases the number of inappropriate pages loaded, which makes it even less likely that *Slicer* will benefit from resident pages when it loads the next slice plane.

To address this problem, we load the data for many planes at once into a three dimensional array. Contiguous sequences of elements are loaded in a single *read()* call. This method has several beneficial effects. First, it reads more data from each filesystem page, 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 array 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 *Slicer* to sometimes take advantage of the filesystem prefetching that is otherwise a liability.

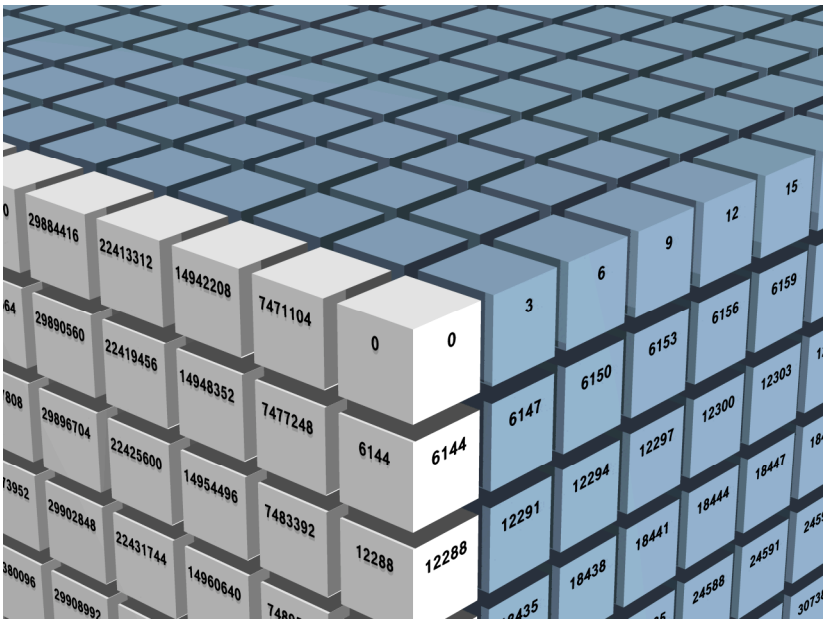


Figure 2. A closeup of the circled corner of figure 1. Numbers indicate offsets in the one dimensional file space. None of the elements in the white slice plane are contiguous, and are all greater than 4K apart from each other.

3. RELATED RESEARCH

Providing efficient access to huge scientific datasets is a challenging problem, and has attracted a lot of attention. A great deal of work has been done in out-of-core visualization algorithms[8-14]. In the operating system and scientific data management communities, work has focused on either providing comprehensive scientific data and metadata management systems, or optimizing file systems using techniques like prefetching, caching and parallel I/O.

3.1 Chunking

Reorganizing datasets on disk to speed access has been explored by a number of researchers. Sarawagi and Stonebraker[15] describe *chunking*, which groups 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[16] 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[17,18]. Cox and Ellsworth[13] compare the use of both chunked and plain files within the context of their application controlled paged segment system.

Chunking is a very effective and general technique, and the Granite system supports chunked file organization. However, 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. In another report, we have shown that the caching strategies adopted by the Granite system result in I/O performance that is competitive with and often superior to chunked files without requiring data reorganization. In addition, these strategies can also be applied to a chunked file to achieve significantly better performance[19].

3.2 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.*[20] describe an algorithm that produces an optimal schedule for prefetching and evicting one dimensional 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.*[21] 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[22]. Cao *et al.* have had success by giving applications control of data cache replacement strategy in their share of cache blocks[23].

When no explicit information about access pattern is available, the history of prior accesses can be used to predict future accesses. Ma keeps tracks of gaps between accesses in order to predict and prefetch the next block of data[24]. Amer et al. group files together based on historical file access patterns[25]. Other researchers have used probability trees or graphs to represent the likelihood of future block accesses given past and current block accesses[26-29]. 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[30,31].

At the application level, Chang[3] adds a separate thread to the user program that performs prefetching by mimicking the I/O behavior of the main thread and preloading data. The VisTools[32] 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. Then, the formatted pages are 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 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. 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 can be inefficient for an n-dimensional access pattern. Figure 3 shows an

example of a column-by-column iteration through a 2D dataset split into pages of 5 elements each. At step 0 of the iteration, the striped page in the upper left of the diagram is loaded into memory. However, the second element in this page is not visited until the iteration has reached step 8. Worse, the last element in this page is not visited until step 32. This means that if we are to use all the data read in the first page, we must keep this page in memory until much later in the iteration. The same argument holds for all the other pages that are loaded as the iteration proceeds down the first column. The size of very large datasets and the pages themselves prohibits all these pages being kept simultaneously in memory. Pages must be discarded before all the data has been used, and then reloaded at a future time. The problem is a result of the one dimensional nature of paging, but a similar argument can be made for chunking when the dataset organization is poorly suited to an unexpected access pattern. See ref. 19 for a more thorough discussion of chunking in the context of the Granite system. The work described in this paper addresses these issues by creating cache blocks that are n-dimensional, and shaped according to the iteration.

0	8	16	24	32	40	48	56
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63

Figure 3. Elements nearby in the numbered iteration sequence are not contained in the same page.

Caching and prefetching methods implemented at the file system level work with little or no explicit information about access pattern. Such algorithms risk prefetching the wrong data, or having to make room in a cache by discarding blocks that will eventually need to be reloaded. However, the approach described in this paper takes advantage of 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.

4. THE MULTIDIMENSIONAL DATA MODEL

We model the data to be processed as a *datasource*. The datasource represents the multidimensional data to the Granite user, and uses a *storage model* to help translate the n-dimensional data space to the one dimensional file space. The storage model can 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[19]. For this paper, we restrict our discussion to files left in their native format.

4.1 Datasources

The datasource is conceptually an n-dimensional array containing a set of sample points. The array indices define the *index space*. Each index space location has a collection of associated data values, called 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 index space location, 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 n-dimensional array of datums.

4.2 The Rod Storage Model

While a datasource has an index space that is n-dimensional, the file is a one dimensional entity. 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 a kind of *axis ordering* called the *storage ordering*. An axis ordering is simply a ranking of axes from *outermost* to *innermost*. "Innermost" and "outermost" suggest position in a set of nested *for* loops. The innermost axis changes most frequently and is called the *rod* axis when referring to storage orderings. Axes are labeled with numbers, so an axis ordering is really just a list of integers. For example, the storage

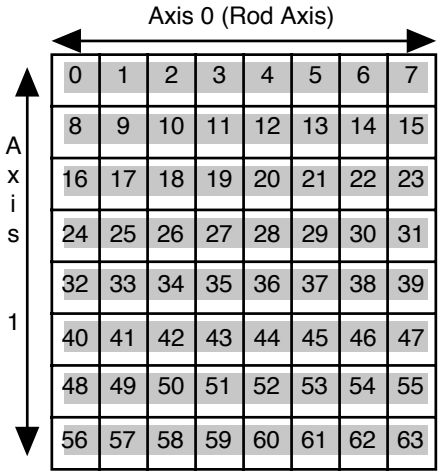


Figure 4. The numbers indicate the ordering of elements in the one dimensional file. The storage ordering here is {0,1}, and the shaded regions indicate the rods for the file.

ordering for figure 4 would be {0,1} if axis 0 is vertical and axis 1 is horizontal.

I/O performance depends on the number of separate read requests made to the storage device. It is important to minimize the number of reads from disk 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 the index space as well as the file space. Consequently, rods are always aligned with the rod axis. Rods can be accessed in a single read operation. When a subblock query is processed, the requested region of index space is decomposed into a collection of the rod subsets contained entirely within the region. We then retrieve the subblock data from disk in rod-by-rod fashion where each rod is read with a single I/O operation. In the case where a set of rods is itself contiguous (or nearly so) in the file, we issue only one read and retrieve many rods in one disk operation.

It is important to note that the rod storage model is a conceptual view of an n-dimensional dataset stored in a one dimensional file. It does not require any reordering or reformatting of the data on disk. The main function of this model is to provide a conceptual foundation for the prefetching technique described in section 6.

5. 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 invocation of the iterator's *next()* method. This value might denote a single location in the index space, or perhaps an entire region. In either case, the iterator value can be used directly in both datum and subblock queries.

The pattern of iteration is determined when the iterator is constructed by the application built on top of Granite. At this time, we have implemented a variety of iterators that explicitly define the complete iteration pattern. It would be straightforward, however, to implement *imprecise* iterators in which some or all of the iteration pattern is chosen by the iterator itself. For example, if the application doesn't require any special order at all, an imprecise iterator would choose to access data in the storage order for maximum performance. If the application only specifies that it wants to access data in a slice by slice fashion, an imprecise iterator would be free to choose the best access pattern for accessing data within each slice.

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. Along with the axis ordering, all this information is useful and available when the system creates a prefetching cache tuned to the iteration.

6. CACHING AND PREFETCHING

Caching algorithms help to accelerate performance by loading a subset of data from a large, slow store into a smaller, faster store. For our purposes, the small, fast store is memory, and the larger, slower store is a disk.

Most cache methods view files as one dimensional entities, but this view of the data is not adequate for scientific applications involving multidimensional datasets because it misses the neighborhood relationships that the user needs.

The problem becomes even more acute as the dimensionality of the dataset increases. To address this issue we have designed a *multidimensional cache* that corresponds to the user's dimensional view of the data.

In this section we present a brief conceptual overview of the caching and prefetching employed in the Granite system. For a more thorough and formal discussion, see ref. 19.

6.1 Multidimensional Cache Blocks

Because our multidimensional cache model is aimed at supporting multidimensional array data, we organize the cache itself as a collection of data blocks, called *cache blocks,* with the same dimensionality as the data. A significant component of the caching strategy is to determine how to shape the cache blocks to most effectively improve I/O performance.

6.2 Choosing Cache Block Shape

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 discarded. We call such a cache block shape *well formed* with respect to the iteration. Caches with blocks well formed for an iteration do not reload discarded blocks when the iteration is performed.

Generally, cache blocks with long extent in the iterator's run axis will increase performance by effectively prefetching data that will soon be needed by the iteration. We refer to this notion as *spatial prefetching*. Unlike other prefetching methods, spatial prefetching minimizes I/O costs by reducing the number of reads made to disk.

Producing an effective cache is easiest when the iterator run axis is the same as the datasource rod axis, as shown in Figure 5a. In this case, simply reading an entire iterator run, or even a portion of a run, greatly accelerates I/O performance. Because the rod and run axes are aligned, we can fill the entire cache block with a single read, yet we are assured that the entire cache block will be used by the access pattern. In this situation, even an operating system cache that is not aware of the data dimensionality performs well.

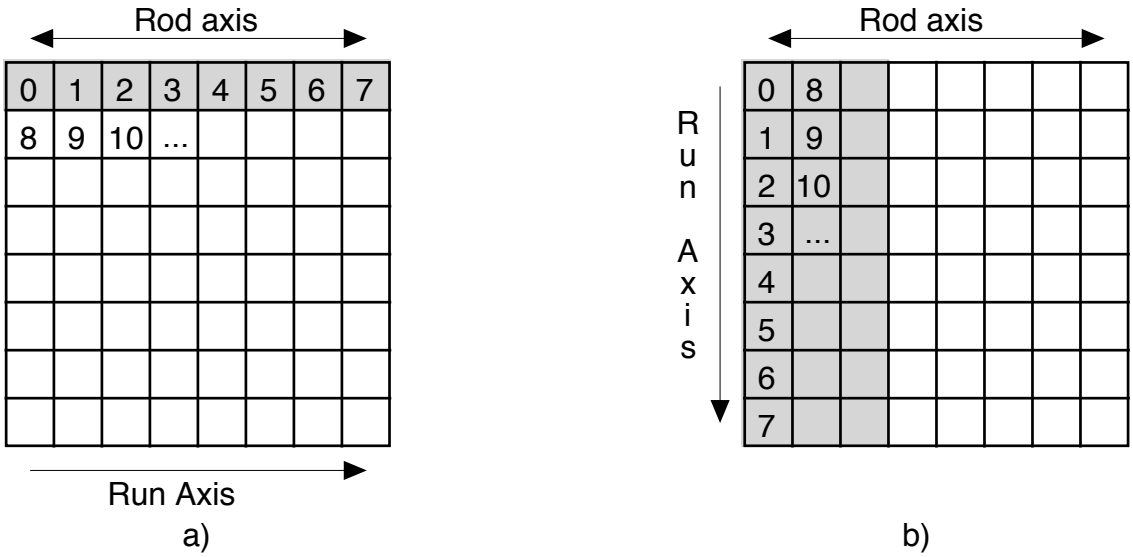


Figure 5. a) Both the storage and iterator ordering is {0,1}. The shaded region represents an effective cache block shape. b) The storage ordering is {0,1}, but the iterator ordering is {1,0}. The shaded region represents an effective cache block shape.

Operating system caching is not nearly as helpful when the rod and run axes are not the same, as shown in Figure 5b. The iterator ordering in this figure is {1,0}, which indicates that it traverses the index space in column-by-column fashion. Here, awareness of dimensionality is necessary in order to provide effective speedup. Our approach is to use the axis ordering to guide the shaping of the cache block. Along the run axis, the cache block is given the same extent as the space being iterated over. The same is done for the next most frequently changing axis, if possible. The process continues, proceeding right to left through the axis ordering, until the memory allotted for the cache is exceeded, or the entire iteration space is contained in the cache block. We then check to ensure that the cache block has some reasonable extent along the rod axis. We call this property *practicality*. If a cache block isn't practical, then using this block will result in too many short reads to provide a speedup. We must therefore allocate more memory to cache block construction to provide useful results. After this last check, we have a cache block shape that ensures the contents will be entirely used by the iterator, and will also accelerate I/O performance by reducing the number of disk reads. For a more detailed examination of our cache block shaping algorithm, see ref. 19.

6.3 Threaded Prefetching

Caches that use only spatial prefetching have achieved significant and useful speedups in our tests[19]. These caches only fill a cache block upon receipt of a query, which forces the application to wait while disk access is performed. This can be a particular disadvantage for interactive programs since it can lead to annoying discontinuities in visualization or interaction response. These and other applications can achieve even better results by employing a separate thread to perform the actual I/O operations that prefill the cache block. We want the application to process one block while the I/O thread reads the next block. We call this *threaded prefetching*.

For threaded prefetching to be effective, the iterator must provide advance knowledge of the access pattern. Furthermore, the application must take enough time processing the data in one cache block to allow the I/O thread to make significant progress in preloading the next. Ideally, the I/O thread would be finished by the time the next block is required, but as long as it has made significant progress, the delay in fulfilling the query will be reduced.

Reducing query delay is particularly useful when visualizing data using animation. A human user finds any choppiness in the animation distracting. Delays in disk access are a common cause of such choppiness, and can be mitigated by using threaded prefetching.

The application described in section 7 uses a *threaded slice cache* that is a specialized version of our threaded spatial prefetching cache. The cache blocks for this cache contain 2 or more slices of the iteration space, which simplifies access and slightly reduces cache overhead.

7. *SLICER* PERFORMANCE EVALUATION

The Visible Woman dataset from the National Institute of Health was used for our evaluation. Tests were performed for the three slice axes for two different volume subsets. The storage ordering is always {0,1,2}, since other storage orderings would yield symmetric performance. The results indicate that Granite's prefetching brings interactive viewing of large datasets within the reach of an experimenter using a commodity machine.

7.1 Slicer Implementation

Slicer was implemented in Java 1.4.2 using the *jogl* OpenGL library. Each slice of the volume is rendered by issuing a subblock query to the datasource layer, and then sending the resulting data directly to OpenGL as a texture. OpenGL then applies the texture to a rectangular shape on screen. There is essentially no processing being done on the data itself, except that which is directly related to the I/O. *Slicer* was run on 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 has an average read latency of 3.8ms.

Slicer includes an optional governor mechanism to provide a maximum frame rate for the visualization. This is common with programs that use hardware rendering. The governor evens out any inconsistencies in the frame generation and frame rendering processes and generally provides smoother, more consistent visualizations when used

with threaded prefetching. In addition to governor frame rate, *Slicer* provides user control over the type of cache, cache memory size and the slicing axis.

Because the *Slicer* application is I/O intensive and requires very little computation for the rendering, the performance overhead imposed by Java is not a significant factor in the total run time. This makes it an effective demonstration of the I/O performance improvements that our prefetching method can provide.

7.2 Evaluation Methodology

Linux has a very effective file system cache that loads and stores 4k blocks of data from disk. Of course, if some or all of a file is already in this cache, stalling costs are greatly reduced or eliminated. The file system also prefetches blocks that are stored following a requested block. Such prefetching is based upon a one dimensional view of the file, and can perform poorly with multidimensional datasets, especially when those datasets are far larger than the available RAM.

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 a good thing in general, it is problematic for our testing environment. In order to give valid and consistent performance statistics, each test must be independent of what happened previously. We wrote a small program that effectively "empties" the cache by filling it with blocks from a dummy file that is not used in the tests. This guarantees a consistent environment by always starting with an empty file system cache, and more clearly shows the effectiveness of our own caching.

Even with an empty file system cache, file system prefetching is still active. This effect is most obvious when the iteration pattern matches the file storage pattern. In this case, the file system prefetches the same blocks that our cache strategy identifies for prefetching, so we achieve only modest improvement (if any). For other iteration patterns the effect of file system prefetching is less obvious. It may have a negative effect on performance due to unwarranted reads, but Granite's prefetching is sometimes able to take advantage of prefetched pages.

Note that we have not presented the *hit ratio* metric that is commonly used to measure cache performance. Since the access pattern is known in advance, it is rare that the application requests data that is not already in the cache. When threaded prefetching is used, cache blocks are filled concurrently with application access, so normally 100% of the data requests can be satisfied from cache. If the application consumes data faster than it can be read, or if spatial prefetching is used alone, the hit ratio is almost as high, since cache misses only occur when a cache block has been exhausted, and a new block must be read from disk. For example, the hit ratio for the 32 slice spatial cache in Table 1 when iterating along slice axis 1 is (2048 - 2048/32) / 2048 = 97%.

	Slice Axis			Slice Plane Dimensions	Cache Slices
	0	1	2		
Slice Axis Length	5186	2048	1216		
No Cache	15.9	1.8	1.6	256 x 256	
Spatial	14.3	11.9	12.0		32
Threaded	20.3	10.0	10.2		32
No Cache	10.6	0.89	0.04	512 x 512	
Spatial	11.2	10.3	2.4		128
Threaded	12.4	8.8	2.2		128

Table 1. Frames per second for the plain datasource, spatial prefetching, and threaded prefetching caches.

7.3 Slicer Results

The Visible Woman dataset has dimensions 5186 x 1216 x 2048 with RGB byte values for each location, giving a total size of 39GB. We compared performance with no cache, with spatial prefetching, and with threaded prefetching. For

each case, we tried all three principal view directions (slice axes). Sample images showing several closeup views along all three slice axes can be seen in Figure 6 at the end of this paper. Only views along axis 0 are "natural", in that they correspond to photographs of body slices. All other views are synthesized by *Slicer*.

Table 1 shows the maximum frame rates for each of the cases. For these tests, the frame rate governor was turned off, and a stable average frame rate recorded. *Slicer* is able to "wrap around" when it finishes an iteration, but we only recorded frame rates from the first pass, to minimize the effect of the file system cache.

For the first set of tests, the slice had dimensions 256x256, with the remaining dimension set to the extent of the entire data volume. Caches were given enough memory to store 32 slices. The second set of tests displayed slices of dimensions 512x512, with memory for 128 slices given to the multidimensional caches. For axis 0, the performance without our caching is quite good, since file system prefetching is very effective for this access pattern. It is even slightly better than plain spatial prefetching, since it avoids cache overhead costs. However, with the addition of the threaded prefetching we are able to show a small but noticeable improvement.

For the other two orderings, the file system cache is unable to match the performance of either of our two multidimensional caches, which are 5 to 60 times greater than the file system cache alone.

When the slice plane includes the rod axis, as with slice axes 0 and 1, each slice can be read as a series of long rod reads. However, for slice axis 2, each datum in a slice is in a separate rod, which dramatically increases the number of reads. Happily, our multidimensional caches are able to extend the rod length along the slice axis, resulting in the performance improvements shown. The threaded cache performs slightly worse here, because it has two cache blocks of half the size of the cache doing spatial prefetching alone. For axis 2, this means that the rods in the threaded cache are half the size of those used with spatial prefetching which entails twice as many disk reads. Using a monitoring tool to view CPU load, we noticed that CPU load rises to 100% during disk activity for the axis 2 tests, but not for the other directions. This heavy load is likely due to the processing required for each read to disk. This makes it much more difficult for our threaded cache to show an advantage over plain spatial prefetching in this situation, since the application is essentially CPU bound. However, this problem can be overcome if enough memory is available. Using a 512 slice cache with axis 2, we got frame rates of 3.6 for spatial prefetching alone and 4.0 with threaded prefetching.

With the frame rate governor turned off, there is a pause whenever a cache block is exhausted and a new block has to be fetched from disk. The pause is lessened but not entirely avoided with the threaded slice cache, since the rendering process is able to run through a block much faster than the next block can be loaded.

The viewer of an animation is distracted by stops and starts in the motion. With threaded prefetching, setting the frame rate governor to the average frame rate results in smooth animation. This slows the rate at which the renderer runs through a block, so that the next block is ready when it is needed. This situation simulates expected behavior when the Granite system is used with a more heavyweight renderer, such as a splatting based volume renderer[33]. In the optimal situation, a renderer that takes as much time to exhaust a cache block as it takes to load the block will show twice the performance with the threaded cache compared to spatial prefetching alone. Since the next block is ready just when it is needed, performance should be similar to the case where enough RAM is available to hold the entire dataset, even with very large datasets like the Visible Woman.

8. CONCLUSION

The Granite Scientific Database provides an interface for efficient out-of-core access that presents a conceptual view of the data as a single volume. The programmer is therefore spared the considerable effort of choosing how to read data from disk most efficiently. By specifying an iterator, the programmer communicates the access pattern to Granite, which is then able to implement an efficient access strategy.

Our system is particularly valuable to the visualization programmer because I/O is often a bottleneck in visualization, and because very large datasets are becoming increasingly common. Through the use of caching, our approach can put much larger datasets within reach of commodity machines.

With our demonstration application, we have shown that our approach can deliver important improvements in I/O performance to visualization applications. *Slicer* shows a significant increase in frame rate as well much smoother animation quality when our caches are employed.

This paper is focused on comparing our prefetching with file system prefetching. The Granite system also supports access to distributed data via *remote datasources*. We still need to evaluate how effective our caching is for this scenario. We will examine the effectiveness of the Granite data access system with large three dimensional time series data. Since locality of reference declines with dimensionality, we expect to see an even greater advantage compared with file system caching alone. We will also extend our toolkit of iterators and caches to support a wider range of iterations, including slice iteration that is not limited to the principal axes. We also plan to expose a well designed caching interface to the application programmer, allowing them to create their own iterators that take advantage of the Granite caching mechanism. Lastly, the Granite system provides support for unstructured data in the *Lattice* layer, and we plan to apply our multidimensional caching scheme at that level.

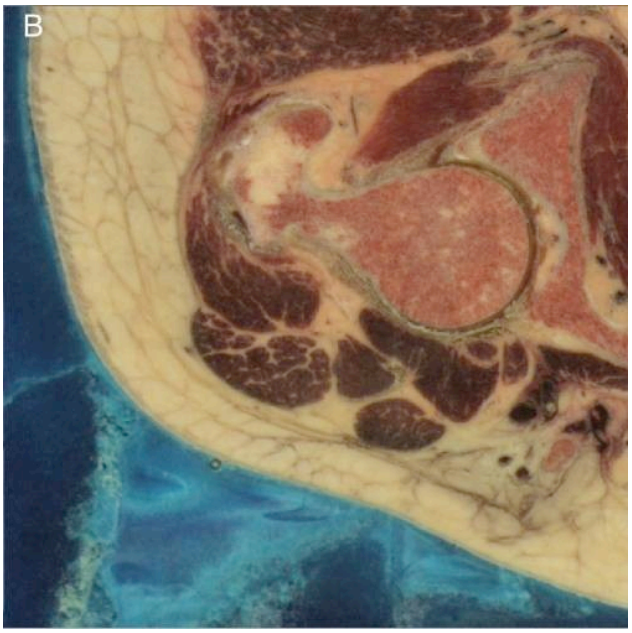
ACKNOWLEDGMENTS

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

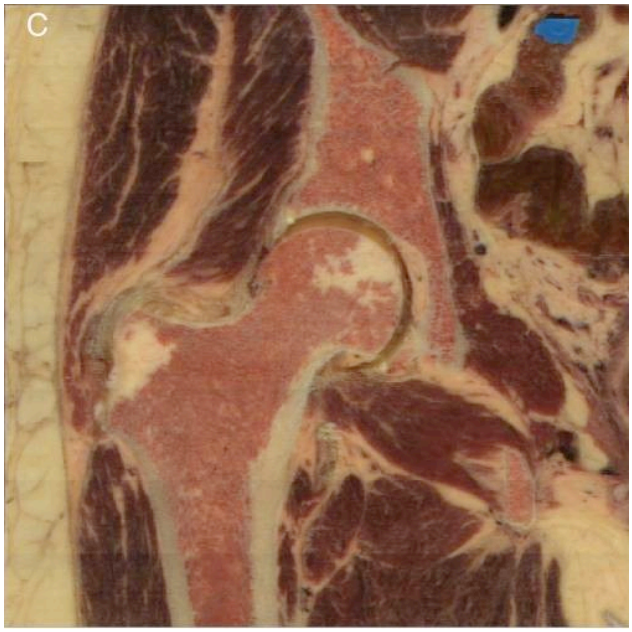
REFERENCES

1. Paolo Cignoni, Claudio Montani, Enrico 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
2. William L. Hibbard, David T. Kao, and Andreas 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
3. F. Chang, Using Speculative Execution to Automatically Hide I/O Latency, *Ph. D. Dissertation*, Carnegie Mellon University, 2001
4. Coughlin, Thomas, High Density Hard Disk Drive Trends in the USA, tech report at http://www.tomcoughlin.com /techpapers.htm
5. Harten, A., Multiresolution Representation and Numerical Algorithms: A Brief Review, *NASA Cont. Rep. 194949,* ICASE, Hampton, VA 1994
6. Rhodes, Philip J., R. Daniel Bergeron, and Ted M. Sparr, A Data Model for Distributed Multisource Scientific Data, *Hierarchical and Geometrical Methods in Scientific Visualization*, Springer-Verlag, Heidelberg, 2001
7. P. Bhaniramka, Y. Demange, OpenGL Volumizer: A Toolkit for High Quality Volume Rendering of Large Data Sets, *Proc. Volume Visualization and Graphics Symposium 2002*.
8. C.L. Bajaj, V. Pascucci, D. Thompson and X.Y. Zhang, Parallel Accelerated Isocontouring for Out-of-Core Visualization, *1999 IEEE Symposium on Parallel Visualization and Graphics*, pp. 97-104, ACM Press, 1999.
9. R. Bruckschen, F. Kuester, B. Hamann and K.I. Joy, Real-time Out-of-Core Visualization of Particle Traces, *2001 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pp 45-50, ACM Press, 2001.
10. Y.J. Chiang, R. Farias, C. Silva and B. Wei, A Unified Infrastructure for Parallel Out-Of-Core Isosurface Extraction and Volume Rendering of Unstructured Grids, *2001 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pp 59-66, ACM Press, 2001.
11. Y.-J. Chiang and C.T. Silva, I/O Optimal Isosurface Extraction, *IEEE Visualization 97*, 293-300, Nov. 1997.
12. Y.-J. Chiang, C.T. Silva and W.J. Schroeder, Interactive Out-Of-Core Isosurface Extraction, *IEEE Visualization 98*, 167-174, Nov. 1998.
13. Michael B. Cox, D. Ellsworth, Application-Controlled Demand Paging for Out-of-Core Visualization, *IEEE Visualization 97,* Nov. 1997
14. Wagner T. Correa and James T. Ellsworth, Visibility-Based Prefetching for Interactive Out-Of-Core Rendering, *Proc. 6th IEEE Symposium on Parallel and Large-Data Visualization and Graphics*
15. S. Sarawagi, M. Stonebraker, Efficient Organizations of Large Multidimensional Arrays, *Proceedings of the Tenth International Conference on Data Engineering*, February 1994
16. Sachin More, Alok Choudhary, Tertiary Storage Organization for Large Multidimensional Datasets, *8th NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies and 17th IEEE Symposium on Mass Storage Systems*, 2000
17. Chialin Chang, Tahsin Kurc, Alan Sussman, Joel Saltz, Optimizing Retrieval and Processing of Multi-dimensional Scientific Datasets, In *Proceedings of the Third Merged IPPS/SPDP Symposiums*. IEEE Computer Society Press, May 2000
18. Chialin Chang, Tahsin Kurc, Alan Sussman, Joel Saltz, *Active Data Repository Software User Manual*, http://www.cs.umd.edu/projects/hpsl/ResearchAreas/ADR-dist/ADR.htm
19. Rhodes,Philip J., Xuan Tang, R. Daniel Bergeron, and Ted M. Sparr, Iteration Aware Prefetching for Large Scientific DataSets, *Technical Report at http://www.cs.unh.edu/~sdb* 2004
20. 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
21. Patterson, R.H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J., Informed Prefetching and Caching. In *Proceedings of the 15th* ACM Symposium on Operating Systems Principles, December 1995, PP. 79-95.
22. Brian C. Forney, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Storage-Aware Caching: Revisiting Caching for Heterogeneous Storage Systems, *First USENIX Conference on File and Storage Technologies (FAST '02)*, Monterey, CA, USA, January 2002.
23. 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
24. Ma, Heng, Remote Transformation and Lattice Manipulation, Master's Thesis, University of New Hampshire, Durham, NH 1992
25. A. Amer, D. Long, and R. Burns. Group-Based Management of Distributed File Caches. In *Proceedings of the 17th International Conference on Distributed Computing Systems,* 2002
26. J. Griffioen and R. Appleton, Reducing File System Latency Using A Predictive Approach, *University of Kentucky Tech. Report #CS247-94*
27. T. Highley, P. Reynolds and V. Vellanki, Absolute Cost-Benefit Analysis for Predictive File Prefetching, *U. Va. Tech. Report #CS200211*
28. T. Highley and P. Reynolds, Marginal Cost-Benefit Analysis for Predictive File Prefetching, *Proceedings of the 41st Anuual ACM Southeast Conference (ACMSE 2003)*, Savannah, GA
29. V. Vellanki and A. Chervenak, A Cost-Benefit Scheme for High Performance Predictive Prefetching, *Proc. of Supercomputing '99*, Nov. 1999
30. Madhyastha, T. M., Elford, C. L., and Reed, D. A, Optimizing Input/Output Using Adaptive File System Policies, In *Fifth NASA Goddard Conference on Mass Storage Systems and Technologies*, September 1996
31. Madhyastha, T. M., and Reed, D. A, Input/Output Access Pattern Classification Using Hidden Markov Models, In *Workshop on Input/Output in Parallel and Distributed Systems*, November 1997, pp. 57-67.
32. David R. Nadeau, An Architecture for Large Multi-Dimensional Data Management, *SDSC White Paper*, http://vistools.npaci.edu/
33. K. Westover, Footprint Evaluation for Volume Rendering, *Computer Graphics, vol. 24*, 1990, pp. 367-376

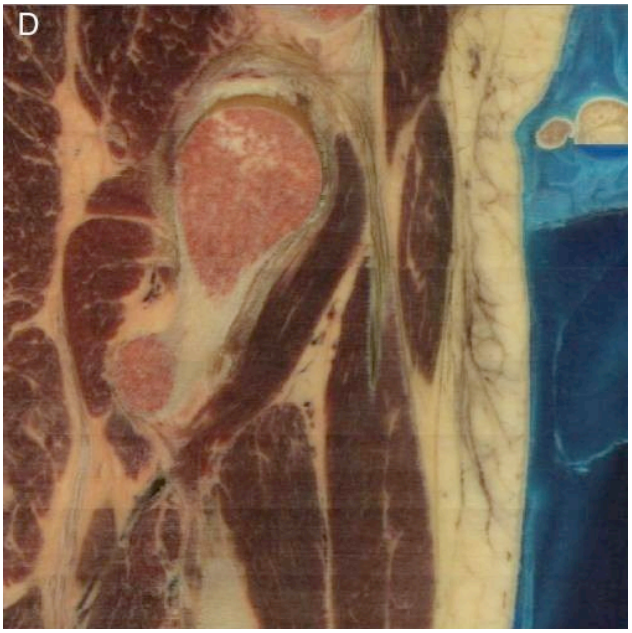
a) a close up view of the waist viewed through axis 1.



b) A view of the right hip joint viewed through axis 0.



c) The hip joint region viewed through axis 1.



d) The hip joint region viewed through axis 2.

Figure 5. Several example images taken from the Visible Woman dataset. All images were produced using a 512x512 slice size. Only 5b is a "natural" image, the other views are synthesized by *Slicer*.