I/O-Conscious Volume Rendering

Chuan-Kai Yang and Tzi-cker Chiueh

Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, USA
emails: {ckyang, chiueh}@cs.sunysb.edu

Abstract. Most existing volume rendering algorithms assume that data sets are memory-resident and thus ignore the performance overhead of disk I/O. While this assumption may be true for high-performance graphics machines, it does not hold for most desktop personal workstations. To minimize the end-to-end volume rendering time, this work re-examines implementation strategies of the ray casting algorithm, taking into account both computation and I/O overheads. Specifically, we developed a data-driven execution model for ray casting that achieves the maximum overlap between rendering computation and disk I/O. Together with other performance optimizations, on a 300-MHz Pentium-II machine, without directional shading, our implementation is able to render a 128x128 greyscale image from a 128x128x128 data set with an average end-to-end delay of 1 second, which is very close to the memory-resident rendering time. With a little modification, this work can also be extended to do out-of-core visualization as well.

1 Introduction

Despite the fact that volumetric data sets are inherently huge, most previous ray casting algorithms research reported performance numbers, assuming that data sets are entire memory-resident. This assumption is not valid when individual data sets are too large to fit into main memory (*out-of-core rendering*), or when users need to browse or explore a large number of data sets. Such assumptions tend not to hold especially on personal workstations, where volume visualization technology is gradually gaining grounds.

The motivation of this work is to develop a high-performance volume rendering system on commodity PCs without special hardware support, with a focus on reducing the *end-to-end* rendering delay, including the disk overhead of bringing the data sets in and out of the host memory. The key technique to minimize the performance impacts of disk I/O is to overlap disk operations with rendering computation so that the disk I/O time is masked as much as possible. To achieve this goal, a volumetric data set is decomposed into blocks, which are stored on disks and accessed as indivisible units. As data blocks are retrieved from disks, rendering computation on those blocks that are brought in earlier proceeds simultaneously. In this execution model, the *minimum* total rendering time for a disk-resident data set is the sum of the rendering time when the data set is entirely memory-resident, and the time required to fetch the first data block.

Surprisingly, the above overlapping execution model is difficult to get right in practice. This paper presents one such optimal execution model: *data-driven block-based*

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

volume rendering, which hides most of the disk I/O delay while at the same time ensures that a data block is completed exercised once it is brought into memory from the disk. The bottom-line result is that on a 300-MHz Pentium-II machine, without directional shading, this implementation strategy is able to complete the task of rendering a 128x128x128 data set into a 128x128 image in 1 second on the average, including the disk I/O time.

The rest of this paper is organized as follows. Section 2 reviews previous volume rendering work that paid attention to disk I/O issues. Section 3 describes the design dimensions of I/O-conscious volume rendering algorithms, and their associated performance tradeoffs. Section 4 proposes a simple extension of this work to do out-of-core visualization as well. Section 5 shows the results of a detailed performance evaluation of the prototype implementation, which is built on top of a Pentium-II machine running Linux. Section 6 concludes this paper with a summary of the major research results. Due to space limitation, we have omitted some details. Please refer to full paper at http://www.ecsl.cs.sunysb.edu/tr/TR89.pdf.

2 Related Work

The main focus of this work is to reduce the disk I/O performance overhead in volume rendering computation, particularly ray casting algorithms. *Out-of-core rendering* refers to the case where the rendering machine's physical memory can not hold the entire data set and thus need to perform disk I/O *during* the rendering process. Cox [4], [3] studied this problem by examining the performance impacts of the operating system interfaces on the disk I/O cost, as well as related file cache management issues. In contrast, our work attempts to use algorithm-specific prefetching to ensure that the data blocks could be brought in before they are needed. The proposed prefetching mechanism is closely tied with the rendering computation, and is completely algorithm-specific. This tightly integrated approach also sets itself apart from other more general-purpose disk prefetching research, as done in [6], [8], [9, 1] and [7]. Another way to reduce the performance overhead due to disk I/O is to use compression to cut down the I/O traffic volume, as done in [10], [5], [2] and [11]. Our work assumes that the ray casting algorithm is more computation-intensive than I/O-intensive, and therefore spending additional decompression computation or restricting the data viewing scope to lower disk traffic is not considered a desirable tradeoff. Rather, we focus on how to *mask* the disk I/O delay.

3 I/O-Conscious Ray Casting Algorithm

3.1 Optimization for Memory-Resident Ray Casting Algorithm

To reduce the end-to-end volume rendering time, the performance of the ray casting algorithm when the data set is completely memory-resident should be optimized to the extent possible. We have added the following performance optimizations to arrive at a high-quality and high-performance ray caster, as the baseline case.

The first optimization replaces floating-point computation with integer arithmetic, specifically in tri-linear interpolations. By replacing the floating-point numbers in tri-linear interpolation, which are between 0.0 and 1.0, with 8-bit integers, we improve the overall performance by almost an order of magnitude in certain cases on a Pentium-II machine, because our ray caster uses only integer arithmetic, and Intel processor's floating-point hardware traditionally lags significantly behind its integer counterpart. This optimization, however, does not affect the rendering quality. For example, Figure 1 and figure 2 show two images rendered through floating-point arithmetic and integer arithmetic without much perceptible differences. The second performance optimization

Fig. 1. Floating point computation.

Fig. 2. Integer computation.

attempts to exploit the instruction-level parallelism using the MMX instruction set extensions available on the Pentium-II processor. MMX is capable of executing multiple low-resolution fixed-point operations in parallel on a high-resolution data-path, e.g., 4 16-bit multiplications on a 64-bit multiplier. By using integer arithmetic and four kinds of MMX instructions: PMULHW, PMULLW, PMADDWD and PSUBW, we create a new version of tri-linear interpolation which takes only 37 instructions. Unfortunately the performance of this code on Pentium-II does not improve much over the non-MMX version, and in some cases actually worsens. Please refer to the full paper for a detailed explanation.

When volumetric data sets are represented as 3D arrays, the address generation logic for the samples used in tri-linear interpolation is susceptible for optimization. Specifically, the eight samples used in tri-linear interpolation have a fixed and simple offset relationship among themselves. By exploiting these relationships to generate the memory addresses of the eight samples involved in tri-linear interpolation, we are able to improve the rendering performance by up to 15%.

The last optimization avenue that we explored is related to caching. We discovered that the ray casting performances for different viewing directions could differ by as much as 30%, although they require the same amount of computation. To improve the cache performance, we have tried to cast a group of rays concurrently rather than one

ray at a time, so that each time a cache block is brought in, it can be utilized as much as possible. However, for reasons as explained in the full paper, the ray group approach does not improve the overall performance. Table 1 shows the performance improvement

Optimization	*Performance Improvement*
Replace Floating-Point with Integer	4 to 6 times faster
Using MMX	0% faster on (Pentium-II)
	60-80% faster on (Pentium)
Hand-Code Address Generation	up to 15%
Caching	No obvious overall improvement

Table 1. *Performance improvements from various optimizations to a generic ray caster implementation on a 300-MHz Pentium-II machine.*

from each of the performance optimizations. For a $128 \times 128 \times 128$ data set with 1-byte voxel and a 128×128 rendered image, the measured ray casting time is 0.68-1.0 sec on a 300-MHz Pentium-II machine. At the same time, the time to retrieve the same data set from the disk is 0.33 sec, assuming that the data set is laid out sequentially. Therefore, it is essential to minimize disk I/O's visible performance overhead to reduce the end-to-end rendering time.

3.2 I/O-Conscious Ray Casting

The general strategy to mask disk I/O delay is to overlap disk I/O with rendering computation. Each volume data set is decomposed into 3D sub-cubes or *macro-voxels*, which are stored contiguously on the disk. However, when a macro-voxel is brought into memory, the voxels are *scattered* into their corresponding positions in the 3D array. In the ideal case, when a macro-voxel is being fetched from the disk, the CPU performs rendering computation on the macro-voxel that is brought in previously, and thus hides all the disk I/O delay. Therefore, the minimum end-to-end rendering time when the input data set is disk-resident is the time to fetch the first macro-voxel plus the time to render the data set when it is completely memory-resident. However, achieving such an ideal overlap between disk I/O and rendering computation remains elusive in practice.

The fundamental mechanism to mask the disk I/O delay is to prefetch the macro-voxels in advance before they are actually needed for ray casting computation. To ensure that the rendering computation should never be stalled due to unavailability of required voxels, the sequence of macro-voxels that are prefetched should be identical to the traversal pattern of rendering computation. In other words, the prefetch stream should traverse the volume data set in exactly the same way as the rays cast. To achieve this effect, the prefetching module should execute the same traversal code as used in the ray caster. Given a macro-voxel size, $B \times B \times B$, it can be shown that as long as the origins of the rays that are cast for prefetching purpose are at most B pixels apart on the image plane, and the sampling distance along these rays remain at 1, then these rays can cover all macro-voxels in the input data set. During prefetching-induced traversal, the algorithm checks whether each sample on each ray steps into a new macro-voxel. If

so, the algorithm brings in the new macro-voxel from the disk; otherwise it continues sampling along the ray.

In summary, the I/O-conscious ray casting algorithm consists of two modules, one for casting rays and the other for prefetching macro-voxels according to the way rays are cast into the input volume data sets. There are three dimensions along which one can implement these two modules. The Cartesian product of the alternatives along each dimension constitutes the entire design space.

Software Structure Because the ray casting module is data-dependent on the prefetching module, careful scheduling between these two modules is essential to mask the disk I/O delay. The current implementation uses the two-thread approach because switching between these two threads incurs a fixed but small thread-level context switch overhead, compared to the two processes approach.

Volume Traversal Strategy The ray casting module can either shoot one ray at a time or a group of rays concurrently. As more rays are cast simultaneously, more states are required to maintain the progress of each ray, and the accumulated color and/or opacity values. On the other hand, the ray group approach enables more processing parallelism in that as the number of concurrently cast rays increases, the CPU is less likely to be idle for the lack of useful work to do. Unlike the CPU cache case, the overhead of state maintenance is well worth the benefits it brings. Therefore, the ray group approach is chosen in the current implementation.

Control Flow There are two ways to pass control between the prefetch and ray casting modules. The traditional approach is *program-driven*, which views the ray casting module as the dominating entity that assumes control most of the time, and occasionally passes control to the prefetch module to bring in the next macro-voxel. This approach requires the system to check each ray in the ray group to see whether the macro-voxel it needs to proceed is available, and if so, advances the ray as far as it can, and then repeats the cycle. When the entire ray group stops, the ray casting module yields the CPU through busy-waiting, until the next macro-voxel is brought into memory. The other approach for passing control is the *data-driven* approach, which advances each ray exactly the same way as the previous approach, but attaches the ray to the macro-voxel that it is waiting for when it stops. Every time a macro-voxel arrives, the system continues the processing for the set of rays that are previously attached to this macro-voxel. The main performance advantage of the *data-driven* approach is that it allows the use of larger ray groups, which improve the processing parallelism, without incurring excessive synchronization checks, which will be the case for the *program-driven* approach. Our current implementation thus chooses the *data-driven* approach for control flow transfer.

Given these design decisions, the I/O-conscious ray casting algorithm works as follows. The prefetch and ray casting modules are implemented as separate threads. The prefetch thread traverses the volume data sets in exactly the same way as the ray casting thread, except that the adjacent rays it shoots are B pixels apart, where B is the dimension of the macro-voxel. The ray group size is the same as the size of the image plane. That is, the ray casting thread starts with as many rays as there are pixels on the image plane. Each ray is initially attached to the first macro-voxel that it encounters while traversing through the volume data set. As the prefetch thread traverses the input data set, it fetches from the disk macro-voxels that have not been brought into memo-

ry previously. Every time a macro-voxel arrives, the ray casting module continues the rays that are currently attached to the macro-voxel. Each such ray will advance as far as possible, until it runs into another macro-voxel that is not resident in memory, at which point the ray is attached to the missing macro-voxel, or it runs to completion.

Figure 3 illustrates this process assuming a 2D data set and a 1D image plane. The prefetch thread shoots only rays in circles whereas the ray casting thread shoots every ray. When the 1-st ray, initiated by the ray casting thread, reaches the 1-st macro-voxel, it checks whether the macro-voxel is already brought into memory. If yes, it steps through the 1-st macro-voxel along the 1-st ray. Otherwise, the ray casting thread enqueues the state of the 1-st ray to the work queue of the 1-st macro-voxel. Figure 3 shows the content of each macro-voxel's work queue when each ray first touches the volume data set boundary. In this case, when the 2-nd macro-voxel is loaded into memory, Ray 3, 4, 5 and 6 will be dequeued in that order and proceed as far as possible until they reach another macro-voxel that is not memory-resident.

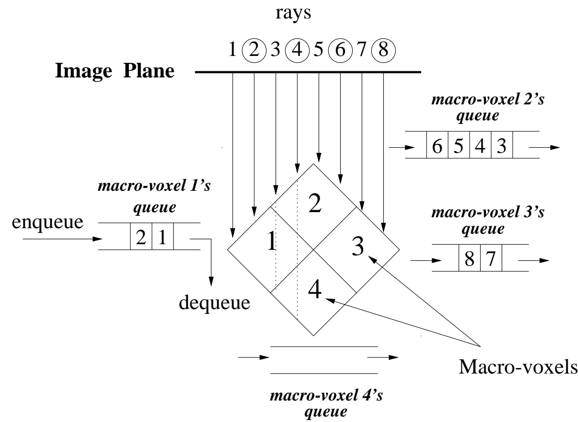

Fig. 3. *A data-driven rendering.*

4 Extension to Out-of-Core Rendering

Because the ray group size is the entire image plane, this means that whenever a macro-voxel is brought in, *all* the rays that need this macro-voxel to advance will be processed before the next macro-voxel arrives. This ray processing pattern leads to two important advantages. First, it exposes the maximum amount of parallelism by identifying all possible rays that are ready to continue. Second, it makes it possible to use a simple FIFO replacement policy for macro-voxels in the case of out-of-core rendering, because once a macro-voxel is "touched," it is no longer needed in future ray processing. For the macro-voxel access pattern to be truly FIFO-like, macro-voxels need to be overlapped with each other by 1 voxel to ensure that each macro-voxel is self-contained during tri-linear interpolations even for rays that pass through the boundaries. That is, a $K \times K \times K$ logical macro-voxel actually contains $(K+2) \times (K+2) \times (K+2)$ voxels

physically. However, in general, the access pattern to macro-voxels is not always FIFO-like, because some macro-voxels that are brought in earlier may be partially blocked by others that are scheduled to be fetched in later. Consider ray 4 in Figure 3. If the first macro-voxel brought in is macro-voxel 1, then because macro-voxel 2 that ray 4 needs is still not in the memory, macro-voxel 1 will still be needed for ray 4 after its traversal of macro-voxel 2, thus making the macro-voxel access pattern not FIFO-like. For the macro-voxel access pattern to be truly FIFO-like, the prefetch thread should bring in the macro-voxels according to their distances to the image plane. That is, the closer a macro-voxel is, the earlier it should be brought into memory.

Instead of sorting all the macro-voxels based on their distances to the image plane, we use the same idea as used in *Splatting* where voxel (in our case, macro-voxel) projection order can be pre-determined and there are only a fixed number of orders possible with respect to all viewing directions. Macro-voxels are then dealt with in that order and attached rays are processed/advanced accordingly.

5 Performance Evaluation

We have implemented a prototype ray caster that incorporates various I/O-conscious performance optimizations described in the previous section. All the following performance measurements are collected from a 300-MHz Pentium-II machine, except those for application-specific file prefetching. The shading model we used is post-shading model, i.e., only density values are interpolated during ray traversal, and then mapped to color and opacity values. We applied linear color and opacity transfer functions and mapped the density value range [0,max] to opacity value range [0,1], where max is the maximal density value. Only grey-scale images are generated and no directional shading is performed.

To overlap disk I/O with rendering computation, volume data sets should be brought into memory incrementally in smaller units, i.e., macro-voxels. Every time one macro-voxel of the input data is available, rendering computation based on this macro-voxel can proceed immediately, presumably in parallel with the disk access for the next macro-voxel. Although smaller disk access granularity facilitates the exploitation of parallelism between CPU and I/O, it has an undesirable effect: the disk access efficiency may suffer because a single sequential disk read of an input data set is now decomposed into a sequence of disk reads, one for each macro-voxel. On the other hand, when CPU processing and disk I/O are fully overlapped, larger macro-voxel increases the start-up overhead, or the time to bring in the first voxel. In the extreme case, the macro-voxel is of the same size of the entire data set, which degenerates into conventional "load and render" approach.

Table 2 shows the loading time measurements for a $128 \times 128 \times 128$ data set under different view angles. We found that $64 \times 64 \times 64$ appears to be the best choice considering both the total I/O time and the start-up overhead. In all the following experiments, we assume $64 \times 64 \times 64$ macro-voxels. Smaller macro-voxels do not perform well because their associated disk access patterns tend to cause excessive random disk head movements.

Macro Voxel Size	Orthographic		Non-orthographic	
	0 0 1	1 0 0	1 1 1	0.3 -0.8 0.4
$128 \times 128 \times 128$	0.33(0.33)	0.33(0.33)	0.33(0.33)	0.33(0.33)
$64 \times 64 \times 64$	0.30(0.070)	0.39(0.071)	0.40(0.070)	0.36(0.070)
$32 \times 32 \times 32$	0.30(0.020)	0.37(0.020)	0.60(0.030)	0.79(0.044)
$16 \times 16 \times 16$	0.34(0.039)	0.48(0.042)	3.25(0.037)	3.40(0.039)
$8 \times 8 \times 8$	0.25(0.038)	0.51(0.038)	3.25(0.037)	3.50(0.035)
$4 \times 4 \times 4$	0.28(0.018)	0.93(0.016)	4.20(0.025)	4.90(0.040)

Table 2. *Total time (sec) to load a 2MB data set* ($128 \times 128 \times 128$) *into memory with different macro-voxel sizes. Numbers in parentheses are the start-up overhead.*

To evaluate the performance of the proposed I/O-conscious ray casting algorithm on an end-to-end basis, we measured the rendering times for three data sets using the conventional approach, which loads the entire data set and performs rendering, and using the data-driven ray casting approach. Then we calculate the optimal bound for the data-driven approach, which is the time to load the first macro-voxel and the maximum of the two: the time to render a volume data set assuming it is entirely memory-resident, and the time to load the remaining macro-voxels. The results are shown in Table 3. As the size of the data set increases, the performance difference between the data-driven ray casting algorithm and the conventional ray casting algorithm widens, because the disk I/O cost is playing an increasingly important role.

Table 3 also demonstrates that the current implementation of the data-driven ray casting algorithm is close to the theoretical optimal bound. The performance difference between the current implementation and the optimal bound also decreases as the data set size increases. This discrepancy comes from the prefetch thread's computation, and additional macro-voxel boundary checks and state maintenance overhead during ray traversal.

To understand the performance gain of the proposed I/O-conscious ray casting algorithm as processors get faster, we render only every other pixel on the image plane, to simulate a factor of 4 improvement in rendering computation. The end-to-end delay measurements for three data sets, *CThead*, *Lobster* and *Brain* and for different view angles are shown on the last two rows in Table 3. For large data sets, the performance gain of the proposed approach, compared to the conventional approach, increases because the disk I/O cost becomes more dominant and therefore the ability to mask it is more important to minimize the end-to-end delay.

Table 4 shows the performance comparisons between the data-driven and program-driven approaches for three different data sets, *CThead*, *Lobster* and *Brain*, and for different view angles. In general, the performance difference between the two approaches increases as the viewing direction moves away from the major axes, because the traversal pattern of the prefetching thread tends to differ more from that of the rendering thread. As a result, the program-driven approach is more likely to be delayed because the prefetch thread is less likely to bring in all the macro-voxels in time for the rendering thread.

Table 5 shows how the ray group size affects the total rendering time under different viewing directions. The results show that the rendering performance improves with the

Viewing direction	CThead (2MB) 64 × 64 image		Lobster (4MB) 128 × 128 image		Brain (8MB) 128 × 128 image	
	Conven. /Bound	**Data-driven**	**Conven. /Bound**	**Data-driven**	**Conven. /Bound**	**Data-driven**
0 0 1	1.33/1.10	1.10	2.97/2.43	2.60	5.63/4.36	4.78
1 1 1	1.01/0.75	0.91	2.49/1.90	2.07	4.86/3.59	3.88
0 0 1	0.61/0.33	0.46	1.3/0.79	0.92	2.43/1.33	1.60
1 1 1	0.56/0.33	0.58	1.3/0.80	1.17	3.37/1.33	2.10

Table 3. *Comparison of rendering time (sec) on PII 300MHz between the I/O-conscious data-driven ray casting algorithm, its optimal bound, and the conventional load-and-render ray casting algorithm, for different data sets under different viewing directions.*

Viewing direction	CThead (2MB) 128 × 128 × 128		Lobster (4MB) 256 × 256 × 64		Brain (8MB) 256 × 256 × 128	
	Data-driven	**Prog.-driven**	**Data-driven**	**Prog.-driven**	**Data-driven**	**Prog.-driven**
0 0 1	1.10	1.25	2.33	2.34	4.78	4.80
1 1 1	0.91	1.40	2.07	2.74	3.88	4.98

Table 4. *Rendering time comparison (sec) between the program-driven and data-driven approaches for three data sets under different viewing directions.*

increase in the ray group size. That is, the performance gain from the ability to exploit more parallelism always out-weighs the additional state maintenance overheads as the ray group size increases.

Table 6 shows the rendering times for a $256 \times 256 \times 256$ using the out-of-core rendering algorithm under different viewing directions and different memory capacity. That fact that the rendering times are within 8% of each other demonstrates this algorithm's insensitivity to the main memory size.

Ray group size	0 0 1	1 1 1
128 × 128	1.10	0.99
64 × 64	1.31	1.15
32 × 32	1.42	1.23
16 × 16	1.46	1.23

Table 5. *Rendering time for a* $128 \times 128 \times 128$ *data set with different viewing directions and different ray group sizes.*

Memory capacity	0 0 1	1 1 1
1 MB	8.74	8.02
2 MB	8.80	8.09
4 MB	8.90	8.22
8 MB	9.10	8.67
16 MB	8.80	8.64

Table 6. *Rendering times for a* $256 \times 256 \times 256$ *data set with different viewing directions and different amounts of memories.*

6 Conclusion

In this paper, we studied the problem of hiding disk I/O delay associated with large-scale volume data set rendering. We attacked this problem by considering in two steps:

make the rendering as fast as possible assuming the data set is already memory resident; mask the I/O latency as much as possible by taking data loading overhead into account. We tackle the former part of the problem by (1) approximating floating-point computation with integer arithmetic without causing perceptible loss of quality on the generated images; (2) speeding up the address generation for the eight voxels used in tri-linear interpolation by exploiting the fixed relationships among them; and (3) employing M-MX instructions to execute multiple instructions simultaneously. To effectively mask the I/O delay, one has to overlap the disk accesses with rendering computation. Data sets are divided into "sub-blocks" or "macro-voxels" to allow separate rendering and I/O threads to work on different macro-voxels. To hide the disk I/O delay, the prefetch thread should preceed the rendering thread for each macro-voxel accessed. We have developed an innovative data-driven approach to exploit as much parallelism as possible while at the same time reducing unnecessary synchronizations checks to the minimum. By incorporating all these optimizations, given a $128 \times 128 \times 128 \times 1$(bytes) data set, our system is able to render a 128×128 grey-scale image in one second on the average using a Pentium II 300MHz machine. For larger data sets, the rendering time scales proportionally. Moreover, we found our system not only can mask the I/O overheads effectively, but also can perform out-or-core rendering effectively without much modification.

References

1. P. Cao, E. W. Felten, A. Karlin, and K. Li. A study of integrated prefetching and caching strategies. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1995.
2. Tzi-Cker Chiueh, Chuan-Kai Yang, Taosong He, H. Pfister, and A. Kaufman. Integrated volume compression and visualization. *Visualization '97*, pages 329–336, October 1997.
3. M. Cox. Managing big data for scientific visualization. *ACM SIGGRAPH '98 Course*, August 1997.
4. M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. *Visualization '97*, pages 235–244, October 1997.
5. J. Fowler and R. Yagel. Lossless compression of volume data. In *Proceedings of Visualization '94*, pages 43–50, October 1994.
6. D. Kotz and Carla Schlattr Ellis. Practical prefetching techniques for parallel file systems. *First International Conference on Parallel and Distributed Information Systems*, December 1991.
7. Tulika Mitra, Chuan-Kai Yang, and Tzi-Cker Chiueh. Application-specific file prefetching for multimedia programs. In *IEEE Multimedia 2000*, July 2000.
8. Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. *The Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.
9. R. H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. *15th ACM Symposium on Operating System Principle*, December 1995.
10. A. Trott, R. Moorhead, and J. McGinley. Wavelets applied to lossless compression and progressive transmission of floating point data in 3-d curvilinear grids. *Visualization '96*, pages 385–388, October 1996.
11. S. K. Ueng, K. Siborski, and K. L. Ma. Out-of-core streamline visualization on large unstructured meshes. *ICASE Report*, April 1997.