I/O-Conscious Volume Rendering

Chuan-Kai Yang Tzi-cker Chiueh

State University of New York at Stony Brook *

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 grey-scale 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. To explore the feasibility of automatically converting memory-resident algorithms into I/O-conscious ones, this paper presents an application-specific file system that transparently maximizes the overlap between disk I/O and computation without requiring application modifications.

1 Introduction

Volume rendering takes a volumetric data set and generates a 2D image. One of the most prevalent volume rendering algorithms is ray casting, which shoots imaginary rays through the data sets and accumulates the contributions of voxel data along each ray according to color and opacity mappings from raw data values. 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.

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

Surprisingly, the above overlapping execution model is difficult to get right in practice. This paper documents the process through which we arrive at what we believe the optimal incarnation of this execution model: *data-driven block-based 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 presents a file system that attempts to automatically overlap disk I/O with algorithm computation for any given program. Section 6 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 7 concludes this paper with a summary of the major research results, and a brief outline of on-going work.

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 [CE97, Cox97] 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. In predictive prefetching [KE91], the system tries to "guess" (through interpolation, for example) the future disk accesses based on the past access pattern observed at run time. Compiler directed I/O[MLG92], on the other hand, analyzes the program, and tries to insert I/O prefetching instructions without getting hints from the application programmers. Application-controlled prefetching [PGG$^+$95, CFKL95] provides procedural interfaces that allows the application to tell, explicitly or implicitly, the underlying file system to retrieve data beforehand. We have also developed a file system that supports application-specific prefetching [MY00], which is to be described in this paper briefly. The major difference between this file system and others is that it supports application-specific disk prefetching without requiring programmer involvement. Given a program, it is capable of generating a prefetch thread that is scheduled to run ahead of the original program thread, to ensure that data are fetched into memory because they are needed.

Figure 1: *A ray-cast image of the head data set, using floating point computation.*

Figure 2: *A ray-cast image of the same data set but using integer computation.*

One way to reduce the performance overhead due to disk I/O is to use compression to cut down the I/O traffic volume. Wavelet-based [TMM96] and DPCM-based [FY94] algorithms have been developed to compress volume data sets in a lossless fashion. In these cases, compressed volume data sets need to be decompressed before being rendered. Chiueh et al. [CYH+97] described a technique to integrate lossy compression and volume rendering in a unified framework, so that rendering can be performed directly on compressed data volume. Another way is to identify the parts of interest on disk first and load them into memory only or one at a time, which essentially involves some "segmentation" work [Fun93, USM97]. 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. In ray casting, it is the precision rather than the dynamic range of floating-point arithmetic that is responsible for producing accurate rendering results. However, most raw volumetric data sets come in the fixed-point format, and the color/opacity transfer functions are also table-driven and thus do not require floating-point arithmetic. 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, rendered through floating-point arithmetic, and Figure 2, rendered through integer arithmetic, show no perceptible differences.

The second performance optimization attempts to exploit the instruction-level parallelism using the MMX instruction set extensions available on the Pentium-II processor. In a word, MMX is capable of executing multiple low-resolution fixed-point operations in parallel on a high-resolution datapath, e.g., 4 16-bit multiplications on a 64-bit multiplier. One good candidate for MMX optimization is tri-linear interpolation, which is expressed as follows:

$$
\begin{aligned}
p_x * p_y * p_z * d_1 + (M - p_x) * p_y * p_z * d_2 + \\
p_x * (M - p_y) * p_z * d_3 + (M - p_x) * (M - p_y) * p_z * d_4 + \\
p_x * p_y * p_z * d_5 + (M - p_x) * p_y * (M - p_z) * d_6 + \\
p_x * (M - p_y) * p_z * d_7 + (M - p_x) * (M - p_y) * (M - p_z) * d_8
\end{aligned}
\qquad (1)
$$

Here all variables are integers, and M is the maximal integer value in the representable dynamic range.

MMX instructions operate against 8 MMX registers, each of which is 8 bytes long. The `PMULHW` instruction multiplies four signed words (2 bytes) in the destination MMX register by the four signed words in the source MMX register, and writes the high-order 16 bits of the intermediate results to the destination MMX register. Similarly `PMULLW` does the same but writes the low-order 16 bits. Another useful instruction is `PMADDWD`, which multiples four signed words in the destination MMX register by the four signed words in the source MMX register. The result is two 32-bit doublewords. The two high-order words are summed and stored in the upper doubleword of the destination MMX register. The two low-order words are summed and stored in the lower doubleword of the destination MMX register. The `PSUBW` instruction, which performs four word-level subtractions at the same time, is useful for the $(M - p_*)$ terms above. By using these four instructions, we create a new version of tri-linear interpolation that takes 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. A careful analysis reveals the following effects that explain the surprising under-performance:

- In order to use the MMX instructions, one must put the operands in the MMX registers. Such preparation is through packing/unpacking instructions, which are relatively restricted. As a result, the computational effort associated with "data preparation" is about 90% of the total computation time in our case, thus offsetting the performance gains from MMX.

- The MMX instruction set is still not sufficiently expressive for our purpose. For example, currently there are no MMX instructions that allow to multiply eight 8-bit operands simultaneously, which would have reduced the total number of instructions required for tri-linear interpolation.

- Pentium-II is able to exploit instruction-level parallelism much better than previous generations in the Pentium family, thus reducing the desirability of performing tri-linear interpolation using MMX. As an evidence, the performance of the MMX version of tri-linear interpolation is actually 160% to 170% faster than the non-MMX version on a 200-MHz Pentium processor with MMX support.

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, because of the following two reasons, the ray group approach does not improve the overall performance. First, as volume data sets are stored as 3D arrays, cache blocks do not necessarily correspond to data chunks required when casting a group of neighboring rays. In other words, casting a group of rays simultaneously does not always help in improving the likelihood that a cache block is completely utilized before it is replaced. Second, casting a ray group entails additional storage overhead to keep the track of the progress of each ray, as well as the related state maintenance processing cost. The "house-keeping" work not only requires more memory space, but also more memory accesses and associated address computation work. Table 1 shows the result of a ray group implementation of the conventional ray casting program. Note that if we use a step by step traversal pattern among all the rays in a ray group, the traditional worse case scenario (0 0 1), where you view the volume data parallel to the Z axe, becomes the best case, while the traditional best case (1 0 0), where you view the volume data parallel to the X axe, becomes the worst case. Therefore exploiting cache effect this way cannot improve the performance universally. One can try to shrink the working set size by using smaller ray group, but as shown in the table, we can not gain anything in the sense of improving the worst case.

Table 2 shows the performance improvement 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 subcubes 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 first design alternative is to put these two modules in a single thread within a single process, using the *asynchronous read I/O* facility available in some operating systems, e.g., `aread` on SUNOS and Solaris, for prefetching purpose. Because the disk I/O occurs asynchronously with respect to the requests, the CPU can continue with rendering computation after setting up the disk read accesses appropriately. It is the programmer's responsibility to insert prefetch calls at proper places in the programs, and to check whether asynchronous I/Os are completed and to take proper actions when they are done. In general, programming with asynchronous I/O is considered more complex and thus more error-prone. The other alternative is to implement the prefetch and ray casting modules as two separate threads but in the same process or address space. In this case, it relies on the operating system to schedule these two threads in a way that the prefetching module is able to bring in the macro-voxels before the ray casting module accesses them. Moreover, switching between these two threads incurs a fixed but small thread-level context switch overhead. Because Linux supports kernel threads but not asynchronous disk I/Os, the current implementation uses the two-thread 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, In addition, because the prefetch module fetches one macro-voxel at a time, the ray group approach is in a better position than the one-ray approach to shorten the *live range* of a macro-

Ray group size	0 0 1	0 1 0	1 0 0	1 1 1
128x128	1.38	1.36	1.91	1.34
64x64	0.97	0.98	1.50	1.06
32x32	0.98	0.97	1.28	0.99
16x16	0.91	0.87	1.00	0.86
8x8	0.94	0.89	0.99	0.85
4x4	1.00	0.91	0.96	0.87

Table 1: *The memory resident execution time (in sec) for a 2MB data set with size (*$128 \times 128 \times 128$*) using different raygroup sizes and viewing directions. Each reported value is an average of multiple measurements.*

Optimization	*Performance Improvement*
Replace Floating-Point with Integer	4 to 6 times faster
Using MMX	0% (Pentium-II) and 60-80% (Pentium) faster
Hand-Code Address Generation	up to 15%

Table 2: *Performance improvements from various optimizations to a generic ray caster implementation on a 300-MHz Pentium-II machine.*

voxel, which is the interval between the time when a macro-voxel is brought in and the time when it is accessed last. Smaller live ranges increases the probability that a given physical memory region is reused for different macro-voxels during the rendering process. 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. Fundamentally this performance difference comes from the fact that the *program-driven* approach attempts to match data consumers (ray casting module) against data producers (prefetching module), whereas the *data-driven* approach matches data producers against data consumers. Because consumers become ready only when data becomes available, it is more efficient for data producers to notify consumers to this effect than for consumers to poll for data availability. 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 memory 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-th ray, initiated by the ray casting thread, reaches the 1-th macro-voxel, it checks whether the macro-voxel is already brought into memory. If yes, it steps through the 1-th macro-voxel along the 1-th ray. Otherwise, the ray casting thread enqueues the state of the 1-th ray to the work queue of the 1-th 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-th 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.

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.

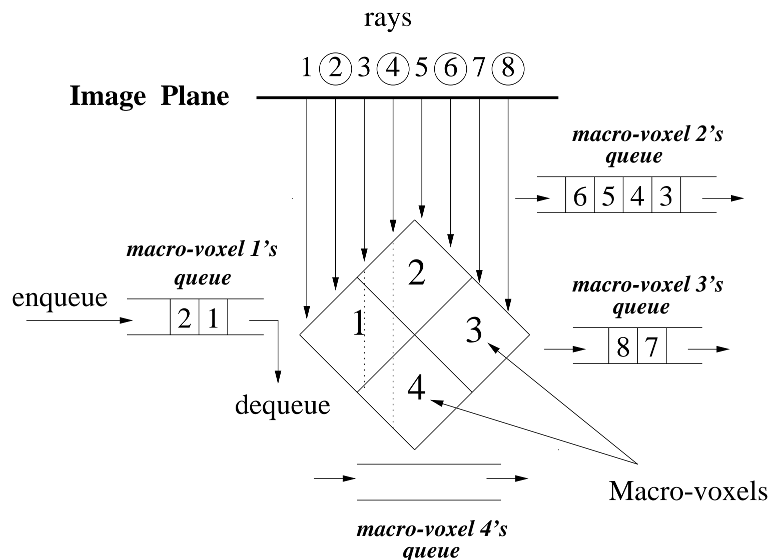

Figure 3: *A data-driven rendering.*

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, the prefetch thread "pre-sorts" all the rays it initiates according to the distance between their corresponding pixels on the image plane and the target data volume, and are organized into a queue. The prefetch thread takes the head entry of the queue out, traverses the next macro-voxel of the associated ray, checks if the ray reaches its end, and puts the entry back to the tail of the queue if the ray can still go on. As a result, the prefetch thread traverses the macro-voxels in a breadth-first and pyramid-like fashion, starting with the one that is closest to the image plane. The ray pre-sorting overhead may be significant, but could be overlapped with the time to fetch the closest macro-voxel from disk and is thus masked.

However, the issue of how to identify the closest macro-voxel without ray sorting still remains. Fortunately, it can be shown that the closest macro-voxel to a given image plane must be one of the eight corner macro-voxels in the data volume. So by comparing the distances between these macro-voxels and the image plane, one can locate the closest macro-voxel, and brings it in while performing ray pre-sorting. In the current implementation, the rendering thread locates the first macro-voxel and fetches it from the disk. At the same time, the prefetch thread performs ray pre-sorting to determine the macro-voxel traversal order. Orthonormal viewing directions should be special-cased, because in this case multiple closest macro-voxels exist, to avoid mismatches between the macro-voxel choices made by the ray casting and prefetch threads.

5 Application-Specific File Prefetching

Although the I/O-conscious ray casting algorithm successfully masks most of the disk I/O delay, as will be shown in the next section, it takes a great deal of tuning and algorithm-specific knowledge to reach this level of performance. One may need to invest the same amount of efforts to make another algorithm I/O conscious. It would be desirable if the principles underlying the I/O-conscious ray casting algorithm could be implemented as a general-purpose operating system facility that overlaps disk I/O and algorithmic computation, so that existing applications can benefit from it *without any modification*. We have developed a prototype file system [MY00] under Linux for application-specific file prefetching (ASFP) that attempts to achieve this goal.

Given an application A, a separate *prefetch* program, P, could be derived, manually or automatically, that includes all the file read statements in P, as well as other program statements related to the computation of file read statements' input arguments. In other words, P is just a subset of A that does nothing but to perform file reads in a *non-blocking* way, playing the same role as the prefetch module with respect to the ray casting module in the previous section. A and P run as distinct threads in the same process. The operating system is modified to schedule P sufficiently ahead of A so that there are enough prefetch calls from P in the disk queue, which ensure that A is not stalled due to file system buffer misses, thus masking the disk I/O delay in most cases.

The key advantage of this approach is that the generation of P from A could be completely automated and thus transparent to the programmers, and between P and the operating system, A's file reads should never incur synchronous disk I/Os because the requested disk blocks have already been prefetched well in advance. However, applying this application-specific file prefetching to a generic ray caster still can not achieve the same level of overall performance as the I/O-conscious ray casting algorithm described in the previous section, because the latter's data-driven computation model exposes more parallelism and reduces unnecessary synchronization check overheads.

6 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 dur-

ing ray traversal, and then mapped to color and opacity values. We applied linear color and opacity tranfer 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.

To understand the tradeoff between disk access efficiency and the start-up overhead, we varied the macro-voxel size and measured the total amount of time required to load a data set into memory. Table 3 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.

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 4. 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 4 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 4. 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.

The program-driven approach insists that it finish the current ray group before starting the next ray group, whereas the data-driven approach, upon the retrieval of a macro-voxel, simply enables whatever rays that are waiting for the macro-voxel. Because the prefetching thread is sampling at a coarser resolution than the rendering thread, the program-driven approach may suffer from additional data waiting overheads due to mismatches between the two threads' traversal patterns. Such waiting would prevent the program-driven approach from continuing with rays in other ray groups, and thus lead to performance loss. Table 5 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.

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 6: *The rendering time for a* $128 \times 128 \times 128$ *data set using the ray group approach for different viewing directions and ray group sizes. Here the macro-voxel size is* $64 \times 64 \times 64$.

Under the data-driven approach, the rendering thread's ray group size should be increased as much as possible to maximize the number of ready rays and thus the amount of CPU parallelism. However, larger ray groups entail more states to be maintained simultaneously, potentially degrading the CPU cache performance. Table 6 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 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.

The goal of application-specific file prefetching (ASFP) is to reap all the performance benefits due to I/O and CPU overlapping without going through a laborious tuning process tailored to individual algorithms. We ran a program-driven ray caster on a Linux system that supports ASFP and on one that does not, and compared their rendering times for different macro-voxel sizes and viewing directions. The results, measured on a 200-MHz PentiumPro machine for a $256 \times 256 \times 256$ data set, are shown in Table 7. The performance difference between ASFP and non-ASFP systems decreases as the macro-voxel size increases, because larger macro-voxel provides most of the prefetching benefits through sequential prefetching. The reason that the performance gain from ASFP is the most when the viewing direction is *0 0 1* is because the data access pattern associated with this viewing direction exhibits the least spatial locality and thus larger macro-voxel does not help much. The last column of Table 7 shows the amount of disk I/O delay that the rendering thread experiences under ASFP, and gives an indication as to how effective the current ASFP implementation is in masking disk I/O delays. ASFP currently uses a static flow control scheme to ensure that the prefetching thread runs sufficiently far ahead of the ray casting thread without overflowing the buffer cache. This scheme works reasonably well with orthonormal viewing directions. However, in the case of non-orthonormal directions, the disk I/O time may increase substantially, and thus lead to buffer underflows, which stall the ray casting thread.

The effectiveness of out-of-core rendering is best evaluated by varying the amount of main memory available on a machine and measuring the corresponding rendering performance. We simulated machines with a different amount of memory by artificially restricting the amount of memory available to the rendering pro-

Macro Voxel Size	Orthonormal		Non-orthonormal	
	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)	1.48(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 3: *The total time (in sec) to load a 2MB data set* ($128 \times 128 \times 128$) *into memory using different macro-voxel sizes. Each reported value is an average of multiple measurements. The numbers in parentheses are the start-up overhead.*

Viewing Direction	CThead (2MB) 64×64 image		Lobster (4MB) 128×128 image		Brain (8MB) 128×128 image	
	Conventional /Bound	Data-Driven	Conventional /Bound	Data-Driven	Conventional /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 4: *The comparison of rendering time (in secs) 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. Measurements are made on a 300-MHz Pentium-II machine, assuming a* $64 \times 64 \times 64$ *macro-voxel size.*

Viewing direction	CThead (2MB) $128 \times 128 \times 128$		Lobster (4MB) $256 \times 256 \times 64$		Brain (8MB) $256 \times 256 \times 128$	
	Data-driven	Program-driven	Data-driven	Program-driven	Data-driven	Program-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 5: *The rendering time comparison (in secs) between the program-driven and data-driven approaches for three data sets under different viewing directions.*

Macro-voxel size (Viewing direction)	Without prefetching	With prefetching	I/O Delay
16(0 0 1)	68.95	31.76	7.0
16(1 1 1)	83.05	64.95	42.5
32(0 0 1)	36.87	31.23	7.3
32(1 1 1)	30.99	29.73	12.0

Table 7: *The rendering time (in secs) for a* $256 \times 256 \times 256$ *data set without prefetching and with prefetching (ASFP). The last column shows the amount of disk I/O time that is not masked under ASFP.*

Memory capacity	0 0 1	1 1 1
2 (512KB)	8.73	8.00
4 (1 MB)	8.74	8.02
8 (2 MB)	8.80	8.09
16 (4 MB)	8.90	8.22
32 (4 MB)	9.10	8.67
64 (16 MB)	8.80	8.64

Table 8: *The rendering times for a* $256 \times 256 \times 256$ *data set under different viewing directions, assuming different amounts of memories, in terms of numbers of* $64 \times 64 \times 64$ *macro-voxels and bytes.*

gram. Table 8 shows the rendering times for a $256 \times 256 \times 256$ using the out-of-core rendering algorithm under different viewing directions. That fact that the rendering times are within 8% of each other demonstrates this algorithm's insensitivity to the main memory size. Note that the data set itself takes 16 MBytes, but as mentioned before, to handle some boundary cases all the macro-voxels should overlap each other by 1 voxel in width, which makes the actual data size become about 18MBytes. Fortunately experiments show that the extra disk I/O overhead associated with overlapping is relatively insignificant.

7 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 MMX 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.

Currently, we are exploring the cache effects on the performance of volume renderers. Although preliminary experiments show that the our-of-core rendering implementation may be able to hold data within L2 caches, a detail study on how to apply the idea of masking the disk I/O delay to hide the memory access delay is needed. In addition, some lossy or lossless data compression algorithms may be applied on top of the current I/O-conscious scheme, provided the decompression rate dominates the data loading rate. Furthermore, in this work, we assume the renderer works on regular-grid data sets. Irregular-grid data sets, whose data access pattern is less predictable, requires more research to mask the corresponding disk I/O delay. Finally, systematically extending this work to a parallel computing system with a parallel I/O facility is another research direction that we intend to pursue in the future.

References

[CE97] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. *Visualization '97*, October 1997.

[CFKL95] 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.

[Cox97] M. Cox. Managing big data for scientific visualization. *ACM SIGGRAPH '98 Course*, August 1997.

[CYH+97] 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.

[Fun93] T. A Funkhouser. *Database and Display Algorithms for Interactive Visualization of Architectural Models*. PhD thesis, University of California at Berkeley, 1993.

[FY94] J. Fowler and R. Yagel. Lossless compression of volume data. In *Proceedings of Visualization '94*, pages 43–50, October 1994.

[KE91] D. Kotz and Carla Schlattr Ellis. Practical prefetching techniques for parallel file systems. *First International Conference on Parallel and Distributed Information Systems*, December 1991.

[MLG92] 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.

[MY00] Tulika Mitra and Chuan-Kai Yang. Application-specific file prefetching for multimedia programs. In *IEEE Multimedia 2000*, July 2000.

[PGG+95] 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.

[TMM96] 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.

[USM97] S. K. Ueng, K. Siborski, and K. L. Ma. Out-of-core streamline visualization on large unstructured meshes. *ICASE Report*, April 1997.