Ray Casting on Shared-Memory Architectures

Memory-Hierarchy Considerations in Volume Rendering

Michael E. Palmer and Brian Totty
Inktomi
Stephen Taylor
Syracuse University

To improve ray-casting frame rates on shared-memory architectures, the authors explore memory-hierarchy effects and their interaction with parallel partitioning and load balancing. With their optimizations, a 16-processor Power Challenge machine renders a 1-Gbyte data set at a frame per second—faster than previously reported for a data set this large.

Volume rendering is the process of rendering two-dimensional images from discretized three-dimensional scalar fields. The data to be rendered consist of space-filling, discretized values called *voxels*. Figure 1 shows example images of volume-rendered regular data, generated by our implementations on the Silicon Graphics Power Challenge. Manipulation of the mapping between voxel value and opacity yields views of the interior muscle and bone structure of this human-body data set.[1]

Recent work in concurrent volume rendering[2–5] has identified two major factors that affect performance:

- the selection of an effective parallel-partitioning and load-balancing algorithm, and
- efficient exploitation of the memory hierarchy—that is, minimizing cache misses at each level of the hierarchy.

This article is a detailed analysis of the memory-hierarchy effects in shared-memory architectures of one method of volume rendering—ray casting (see the adjacent sidebar). We studied two parallel-partitioning and dynamic load-balancing algorithms—one object partition and one image partition—exploring trade-offs between their memory-hierarchy performance and the algorithmic optimizations they allow.

Our resulting implementations (along with careful tuning of the ray-advancement kernel for Silicon Graphics' R8000) yield extremely high performance. For a 1-Gbyte female human-body data set, we attain an average frame rate of 1.0 frame per second, at a resolution of 400 pixels × 300 pixels, on a 16-processor Silicon Graphics Power Challenge. This is faster than the literature has previously reported for a data set this large.

1063-6552/98/$10.00 © 1998 IEEE

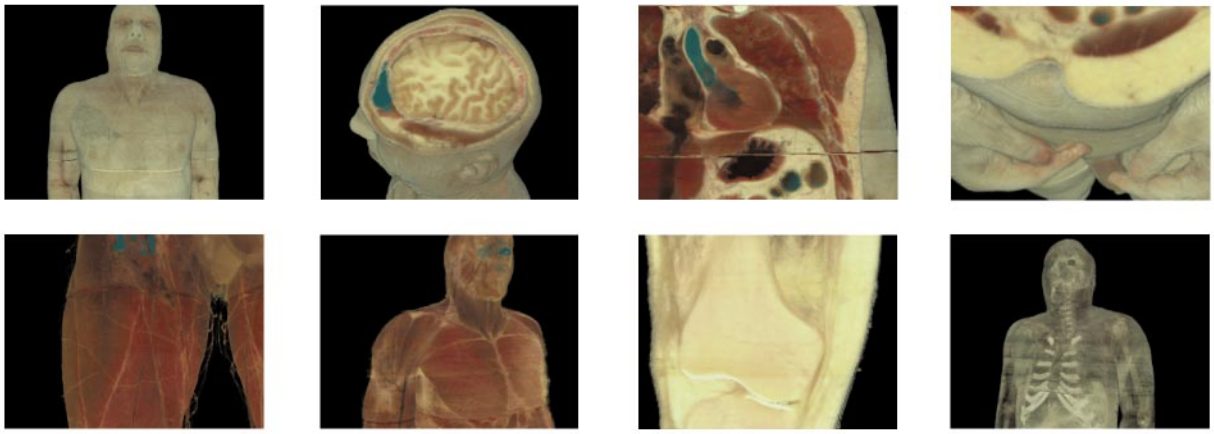

Figure 1. Volume rendering, example output: The ray-casting process, applied to a data set of the male human body, produced these images.

Issues in volume rendering

Many algorithms exist for volume rendering, including projective methods,[1–3] isosurface extraction,[4] shear-warp factorization,[5] and ray tracing and ray casting.[6] This study focuses on a ray-casting implementation for the volume rendering of regular Cartesian data.

RAY CASTING

In the simplest form of ray casting, depicted in Figure A, individual rays originate at the eye point, pass through a pixel of the display screen, and project through a sequence of data elements, or voxels.

We call the coordinate system of the three-dimensional data set *object space*. In Figure A, this set of coordinate axes is labeled x, y, and z. The viewer's coordinate system is called *image space*, and is labeled u (the viewer's right), v (the viewer's up), and n (opposite to the direction in which the viewer is looking).

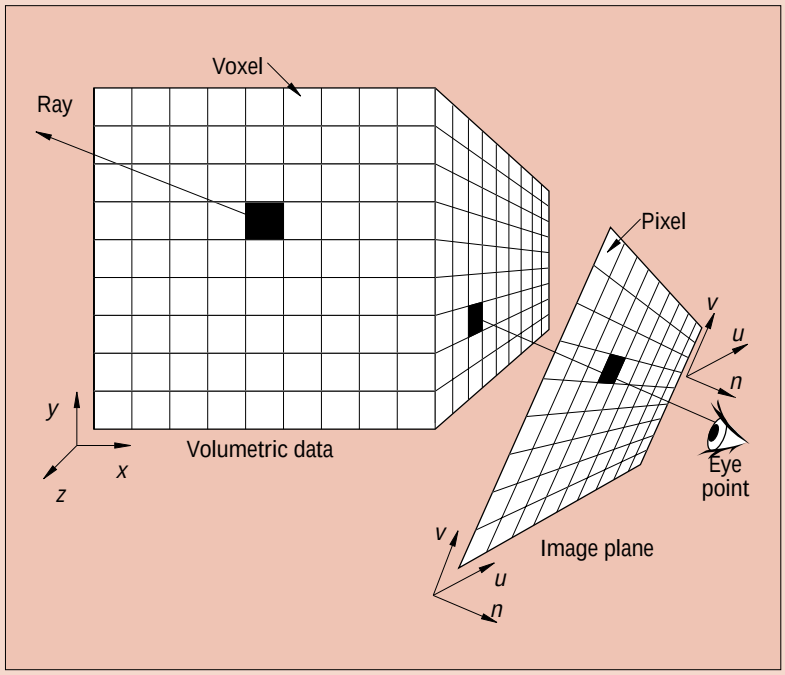

Figure A. Coordinate systems in volume rendering.

As a ray passes through each voxel, it accumulates color and opacity, based on the color and opacity of the voxel and the length of the ray intersection with the object.[7] When a ray exits the volume, the display pixel is assigned the accumulated color of the ray.

A ray's accumulated opacity value asymptotically approaches 1 at a rate that depends on the opacity of the intersected data voxels. A common algorithmic optimization known as *opacity clipping* stops the further advancement of rays that have become nearly opaque, often resulting in a two- to four-fold performance improvement. This optimization is quite dependent on the data set: it works best on data sets with large opaque areas, and not at all on entirely translucent data sets, where rays never become completely opaque.

LAWS OF COMPOSITION

We have some flexibility in the order in which we composite an ordered set of voxels V_1 through V_n to generate the color of a ray. We define the composition operator, C, as

$$ray_{k+1}^{\alpha,R,G,B} = C(ray_{k}^{\alpha,R,G,B}, \; V_{k}^{\alpha R,G,G})$$

This operator is used to take the ray's color (R, G, B) and opacity (α) at step

(Continued on next page)

(Continued from previous page)

k and add to it the effects of the color and opacity of the kth voxel that it intersects. The commonly used composition operator is associative but not commutative. That is,

$$C(C(a, b), c) = C(a, C(b, c))$$

but, in general,

$$C(a, b) \neq C(b, a)$$

Therefore, we need not composite the ordered set of voxels strictly in order. We can composite any contiguous subset of voxels without regard to whether neighboring contiguous subsets have yet been completed. We can then composite the results from these subsets to generate the results for larger contiguous subsets, until ultimately the entire set of voxels V_1 through V_n has been included.

It is this flexibility in the order of composition that lets us manipulate the order of memory accesses to increase memory locality. We will divide the volume data set into subblocks that are stored contiguously in memory, process them one at a time, and then composite these results.

COHERENCE

Blocking the data set to improve memory locality exploits a form of *coherence*.[8] Many algorithmic optimizations to ray casting take advantage of some form of coherence, for example

- *object coherence*—objects tend to be connected, bounded bodies;
- *area coherence*—the 2D projection of a 3D body tends to have a connected, bounded range;
- *frame coherence*—one frame of an animated sequence is likely to resemble the previous frame (for example, in the colors assigned to given pixels or in the costs to render given units of work); and
- *coherence (locality) of memory reference*—a memory reference to a given location is likely be temporally close to references to logically neighboring locations.

References

1. R.A. Drebin, L. Carpenter, and P. Hanrahan, "Volume Rendering," *ACM Computer Graphics*, Vol. 22, No. 4, pp. 65–74, 1988.
2. L. Westover, "Footprint Evaluation for Volume Rendering," *ACM Computer Graphics*, Vol. 24, No. 4, 1990, pp. 367–376.
3. J. Wilhelms and A. Van Gelder, "A Coherent Projection Approach for Direct Volume Rendering," *ACM Computer Graphics*, Vol. 25, No. 4, 1991, pp. 275–284.
4. W.E. Lorensen and H.E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *ACM Computer Graphics*, Vol. 21, No. 4, month, 1988, pp. 163–169.
5. P. Lacroute and M. Levoy, "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation," *Computer Graphics Proc., Ann. Conf. Series*, ACM, New York, 1994, pp. 451–458.
6. P. Sabella, "A Rendering Algorithm for Visualizing 3D Scalar Fields," *ACM Computer Graphics*, Vol. 22, No. 4, 1988, pp. 51–58.
7. J.T. Kajiya and B.P. Von Herzen, "Ray Tracing Volume Densities," *ACM Computer Graphics*, Vol. 18, No. 3, 1984, pp. 165–174.
8. I.E. Sutherland, R.F. Sproull, and R.A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *Computing Surveys*, Vol. 6, No. 1, Mar. 1974, pp. 1–55.

We have also extended our methods to a cluster of such machines. Using eight Silicon Graphics Power Challenge machines with a total of 64 processors, we attain average frame rates up to 10 frames per second on a 357-Mbyte male human-body data set for a sequence of frames generated by interactive user control.

Partitioning strategies

Volume rendering is replete with parallelism. However, selecting parallel-partitioning and load-balancing algorithms that assign tasks and data to processors to maximize load balance while minimizing communication overhead is nontrivial. This has been the focus of a great body of research; Thomas W. Crockett[6] has put together an excellent and recent overview. There are two broad classes of ray-casting partitioning techniques: *image partitioning* and *object partitioning*.

Image-partitioning[5,7] methods assign subdomains of image pixels (for example, lines or rectangles) of the 2D display screen to processors. To render an image subdomain, the processor must access each image pixel and all the voxels of the data volume along the ray that pierces the pixel.

Image partitioning has several appealing attributes. The disposition of a pixel can be solely determined by the owning processor, without compositing contributions from other processors. This allows global opacity clipping (see sidebar). The primary disadvantage of image partitioning is lack of locality in accessing the 3D volumetric data set. To cast one ray, a processor might in general have to access any voxel in the data set. In shared-memory systems, this lack of locality can result in cache thrashing; in distributed-memory systems it can result in large communication volumes. On fine-grain distributed-memory systems, this cost becomes prohibitive, and the image partition is not suitable.

An alternate approach, object partitioning, assigns to processors subdomains of the object space called *blocks* (for example, rectangular solid subsets of the data volume). We use a method similar to the work of Kwan-Liu Ma and colleagues,[8] in which processors cast rays through each block individually, generating a *tile*, the volume-rendered image of a single block. The system then sorts the tiles globally according to their distance from the viewpoint and composites them appropriately to create the final image of the entire data volume. Unlike Ma,[8] we do not use a binary swap among the processors during compositing. This is largely because we use a shared-memory system.

Object-partitioning schemes have a natural ability to improve memory system performance. Because blocks are stored contiguously in memory, processing the set of voxels within a block before processing other voxels dramatically increases locality. For properly sized blocks, orientation-dependent memory system delays vanish, offering the potential of reliable interactive-rendering rates.

Because object partitioning renders blocks individually, however, global opacity clipping is infeasible. (Intrablock opacity clipping is still possible, but it is not as effective.) In addition, there is overhead associated with the algorithm's compositing phase. This phase is parallelizable, but it has a cost proportional to the total area of the projected tiles—roughly the cube root of the number of blocks.

Exploiting the memory hierarchy

Though efficient memory-hierarchy exploitation has been identified as important, the problem has not been completely characterized or solved. In particular, the dependence of memory-hierarchy performance on view direction has not been adequately studied. For example, in our implementation of a fairly standard image partition (which yields high performance for many view directions), memory-hierarchy effects make the fastest view direction 10 times as fast as the slowest. Given the current wide use of interactive volume rendering, consistency of performance independent of viewpoint is particularly important.

We performed our experiments on the Silicon Graphics Power Challenge, a shared-memory multiprocessor containing up to 18 Mips R8000 CPUs clocked at 90 MHz. The processors communicate via a bus with a peak bandwidth of 1.28 Gbytes/s to a shared-memory system. The R8000 has a two-level cache system. The on-chip, direct-mapped L1 cache consists of 512 blocks of 32 bytes for a total of 16 Kbytes; the off-chip, four-way-interleaved L2 cache consists of 32,768 blocks of 128 bytes for a total of 4 Mbytes. The cache system's functions are partitioned by data type: For integer numbers, the L2 cache serves as a streaming cache for the L1 cache; for floating-point numbers, L2 serves as a single-level cache between the processor and memory. We access our data voxels as 1-byte integer values.

We use three tools to characterize the memory-hierarchy performance of each ray-casting algorithm:

- First, we isolate the time that can be attributed to the cache-miss penalty by comparing the ordinary ray-casting algorithm with a modified algorithm that accesses a single, fixed voxel. The modified algorithm eliminates the costs associated with cache misses.
- Second, we use a hardware bus-snooping board to count actual L2 misses.
- Third, we use a software cache simulator that enables us to separate the entire miss-penalty time into L1 and L2 portions.

Ray casting cost analysis

In our system, the inner kernel of the ray caster consumes most of the time required to render a frame. It iteratively advances a ray into the next voxel, reads the voxel's value, maps it to an opacity and color, and adds the opacity and color appropriately to the accumulated opacity and color of the ray. (Our system does not interpolate between voxels.) The read of the voxel's value is the single-memory access that causes the majority of all cache misses during frame generation. In ray casting through regular data, in general, the generation of a single image could require a processor to access any voxel. The distribution of memory accesses greatly affects performance. The viewpoint from which a set of rays are cast and the distribution of the data-set voxels in memory entail a certain distribution of memory accesses; this in turn entails a certain number of L1 and L2 misses.

We can break down the approximate total time per frame into the following parts:

$$\begin{aligned}&\text{total time per frame (viewpoint, number of blocks, number of rays cast)} =\\ &\text{time per intersection (viewpoint, number of blocks)} \times\\ &\text{number of intersections (viewpoint, number of rays cast)} +\\ &a \times \text{number of blocks} +\\ &b \times \text{number of rays cast} + c\end{aligned}$$

where the *time per intersection* and *number of intersections* refer to the intersections of rays with voxels. The time on average to intersect a ray with a single voxel can be separated into

$$\begin{aligned}&\text{time per intersection (viewpoint, number of blocks)} =\\ &d \times \text{L1 misses (viewpoint, number of blocks)} +\\ &e \times \text{L2 misses (viewpoint, number of blocks)} + f\end{aligned}$$

We use a 3D Bresenham algorithm to calculate a ray's

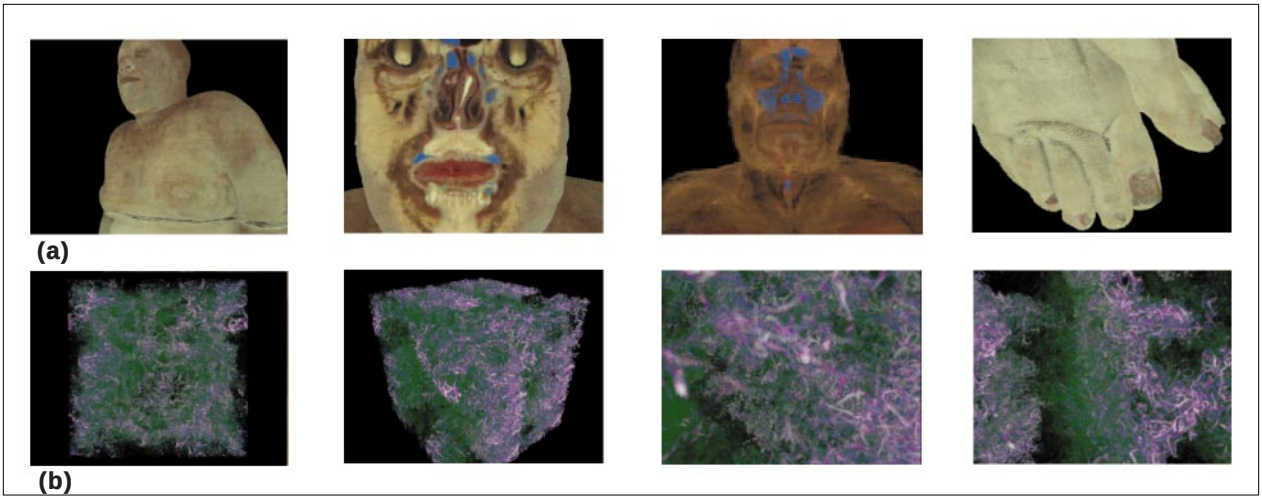

Figure 2. Example images from the (a) female and (b) vorticity data sets.

path. The Bresenham algorithm yields more advancement steps for diagonal rays than for straight rays of the same length (that is, it causes the number of intersections to be a function of viewpoint). Thus, we must apply a division operation to correct for the varying distance of each step. Nevertheless, this was the fastest of several ray advancement strategies we implemented, because we achieved very good software pipelining on the superscalar R8000.

In this article, we do not focus on algorithmic optimizations, such as those intended to minimize the number of intersections. Nor do we explore methods that exploit forms of coherence to reduce the total number of rays cast; we cast one ray for each pixel. Instead, we focus on the trade-off between decreasing the time per intersection (by increasing locality with data blocking) and the overhead resulting from increasing the number of blocks.

The number of ray-voxel intersections in one frame varies with view direction because of several factors. These include the data-set extent in each dimension, effects of diagonal rays in the Bresenham algorithm, and the portion of the data set currently visible. Therefore, a useful, normalized measure of ray-casting speed is the average time per intersection—that is, the average time to advance a ray one step and intersect it with a single voxel.

Our research shows that most of this time for the slowest view directions is attributable to memory-hierarchy effects. The casting of hundreds of thousands of rays through hundreds of millions of voxels places extreme demands on the memory system.

Other researchers[3,5] have not found cache-miss penalties to be the algorithm's main cost component; nor have they identified this strong directional dependence. We suspect two reasons for this: First, we systematically searched the sphere for the worst viewpoint directions—the variation could be easily overlooked. Second, we spent considerable effort optimizing our inner loop for the R8000. This involved using a simple filter (no interpolation between voxels) as well as source-code modifications to allow the compiler to generate efficiently pipelined loops, and it yielded a performance gain of three or four times. This improvement factor was due to more efficient instruction scheduling, not to actual reductions in numbers of memory accesses or to significant changes in the order of memory accesses. We concluded that the cost of memory accesses is the largest remaining cost. Philippe Lacroute[2] pointed to memory-hierarchy costs as a primary cost component, but it was difficult to further improve the locality of the shear-warp factorization algorithm used in that work.

Experimental results

We used three data sets in our experiments:

- a data set derived from the Visible Human Female data set,[1] which was produced by freezing a female cadaver, shaving it into over 5,000 slices, and then digitally photographing each slice;
- a data set derived from the Visible Human Male data set,[1] produced in a similar manner; and
- a computational fluid dynamics data set, courtesy of the Laboratory for Computational Science and Engineering at the University of Minnesota, that displays the vorticity of a simulated turbulent fluid.

Table 1 lists the dimensions of each data set; we stored each at 1 byte per voxel. Figures 1 and 2 show example images of the three data sets. The female and male data sets were produced photographically, so the colors approximate the actual colors of the bodily tissues represented.

As a performance-improving heuristic, opacity clipping is useful for the female and male data sets, because they both contain large opaque objects. However, this is not the case with the vorticity data set, because it is

Table 1. Sizes of experimental data sets.

Data set	Dimensions (voxels)	Total size (Mbytes)
Vorticity	$512 \times 512 \times 512$	128
Male	$584 \times 1{,}878 \times 341$	357
Female	$840 \times 2{,}595 \times 480$	1,000

largely translucent. In addition, the vorticity data set is cubical, which is useful for removing effects due to nonunity aspect ratios when comparing performance between views from different directions.

Test suite

The test suite for our experiments consists of a sequence of 132 views of each data set. We intended this suite to discover the directional dependence of the time per intersection and of the two types of cache miss. Figure 3 shows selected frames from this suite, called axisorbit. The suite comprises four discrete segments, each consisting of the

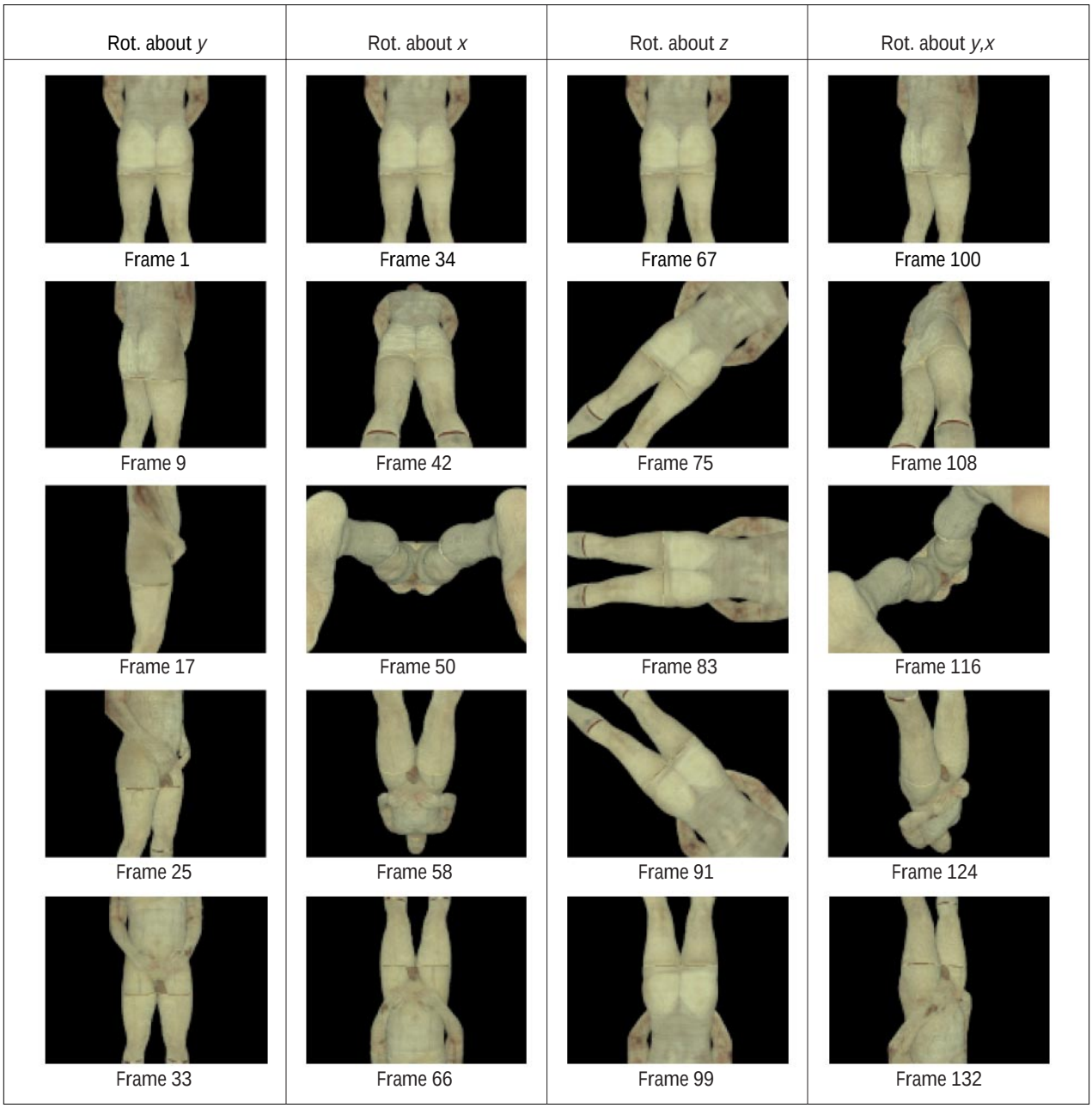

Figure 3. Selected frames from the axisorbit test suite.

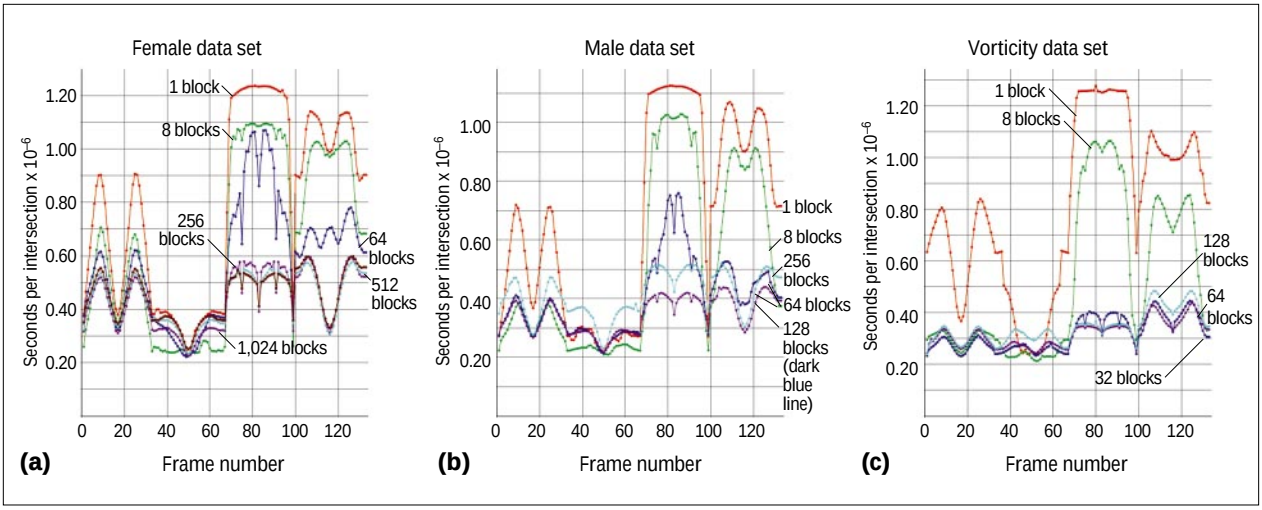

Figure 4. Dependence of time per intersection on view direction for the (a) female, (b) male, and (c) vorticity data sets.

rotation of the data set about a different axis. In frames 1 through 33, the data set is rotated 180 degrees about *y*; in frames 34 through 66 about *x*; in frames 67 through 99 about *z*; and in frames 100 through 132, the data set is initially rotated 45 degrees about *y*, and then from 0 to 180 degrees about *x*. All views are rendered at a resolution of 400 pixels × 300 pixels.

The 3D data sets are linearized in memory by *x* first, then *y*, then *z*. When the cadaver in the female or male data sets is standing upright and facing the viewer, the object space *x*-direction points to the viewer's right, the *y*-direction points up, and the *z*-direction is toward the viewer. This is important to the consideration of cache effects.

Viewpoint dependence of ray-voxel intersection time

We conducted all experiments reported in this section using one processor with opacity clipping turned off. Figure 4 plots frame number versus time per intersection for data sets blocked to different degrees, between 1 and *n* blocks, where *n* is 1,024 for the female data set, 256 for the male data set, and 128 for the vorticity data set. For each data set, we chose the maximum number of blocks *n* to yield minimum-sized blocks of approximately 1 Mbyte, a fraction of the size of the 4-Mbyte L2 cache.

Figure 4 establishes the strong dependence of the time per ray-voxel intersection on view direction. It also shows that we can control this dependence by blocking. The time for a single block (shown in red) shows strong directional dependence for all three data sets. Appropriately sized blocks reduce this dependence: as the number of blocks increases, locality increases, and cache hit rates improve. However, making the blocks smaller past a certain point yields no benefit, and there is an overhead cost associated with increasing numbers of blocks.

For the female data set, the optimal number of blocks (that is, the number that yields the best average frame time for all view directions) is 512 (light blue, Figure 4a). For the male data set, the optimal number is 128 (purple, Figure 4b), and for the vorticity data set, it is 64 (purple, Figure 4c). These numbers yield block sizes of 2 Mbytes, 2.8 Mbytes, and 2 Mbytes for the three data sets, respectively. These sizes fit easily into the 4-Mbyte L2 cache.

By comparing the ordinary algorithm with one that completely flushes the cache between blocks, we determined experimentally that interblock effects are negligible. However, we found that interpixel effects are significant, and are not intuitively obvious. Consider a single large block (red plots in Figure 4). One might assume that viewing directly down the *x*-direction (Figure 3, frame 17) would yield the best cache performance, and therefore the lowest time per intersection. However, this is not the case. Instead, a view directly down the *y*-axis (Figure 3, frame 50) is better. This is due to somewhat complex interpixel effects.

Recall that the data set is linearized in memory *x* first, then *y*, then *z*. As we cast rays, we scan across the display screen in horizontal rows. The R8000 has an L1 cache line length of 32 bytes and an L2 cache line length of 128 bytes. When a ray travels directly down the *x*-axis (Figure 3, frame 17), it intersects voxels in steps of 1 byte through memory. It misses in the L1 cache only once in every 32 ray-voxel intersections; it misses in L2 only once in every 128 intersections. The first intersection that misses fetches into L1 the values for the next 31 intersections, and into L2 the values for the next 127 intersections, yielding good cache performance. However, the next ray uses none of these cached voxels; it starts one pixel to the right of the last one on the display screen.

In contrast, when a ray is traveling directly down the *y*-axis (Figure 3, frame 50), the first ray misses in the

cache at every ray-voxel intersection. It is traveling through memory in steps equal to the *x* dimension of the data set, which is greater than both 32 and 128 bytes for all three data sets. However, it leaves behind in the cache an entire *x*-*y* plane of voxel values in the L2 cache. The dimensions of this plane are the entire *y* dimension of the data set by the length of a cache line. Because the L1 cache consists of 512 lines of 32 bytes, and the data sets each have a *y* dimension of 512 or greater, it is likely that casting a single ray down the *y* direction completely flushes the L1 cache for all three data sets.

However, such a ray does not flush the L2 cache completely. The next ray (one pixel to the right on the display screen) is translated from the last ray by one voxel in the *x* direction. In the approximation that it travels straight down the *y*-axis (not exactly true with perspective projection), it lies entirely in the same *x*-*y* plane of voxels the last ray fetched, so it does not miss in the L2 cache at all—nor do the next 126 rays. Frame 116 (Figure 3) is just 45 degrees rotated from frame 50, but it gets far worse cache performance because rays that are adjacent on the screen do not share an *x*-*y* plane of voxels.

A ray traveling directly down the *z*-axis misses in cache every time, but leaves an *x*-*z* plane of voxel values behind in the L2 cache. In this context, there are two interesting cases in our test suite. If the next ray cast is translated from the last ray by one voxel in the *x* direction (as in frames 1, 33, 34, 66, 67, and 99), it hits cache every time. This yields quite good performance—nearly as good as the best case of frame 50 for the female and male data sets. However, if the next ray cast is translated from the last ray by one voxel in *y* (as in frame 83), it misses every time. Furthermore, by the time the ray one row down on the screen (which would intersect the same *x*-*z* plane of voxels) is cast, intervening cache activity has already flushed the plane from the cache. This is the worst possible cache performance; every ray misses at every intersection. However, we could turn the worst-case performance of frame 83 into the very good performance of frame 1 (nearly the best case for the male and female data sets) simply by scanning across the display in columns instead of by rows.

Analysis of the cache behavior for perspective projection rays traveling in directions not parallel to the coordinate axes becomes so complex as to require a cache simulator, which we discuss in a later section.

Experimental fixed-voxel access

To determine how much of the total frame time is due exclusively to memory-hierarchy effects rather than other overheads, we modified the ray-casting kernel. We replaced the access to the currently intersected voxel with an access to a single, fixed voxel. Of course, this does not generate proper images of the data set, but we intended it to eliminate the costs associated with cache miss penalties. After the first access to this voxel, subsequent accesses result in no cache misses. We call the modified algorithm the *no-memory* algorithm. Figure 5a shows the result for a single block for all three data sets. The original algorithm's time per intersection, labeled *with-memory*, is plotted in red. The time per intersection of the no-memory algorithm is plotted in green; this is the time attributed to everything but memory-hierarchy effects. The blue plot, labeled *memory-only*, is the difference between the with-memory and no-memory lines. This is the time per intersection that we attribute directly to memory-hierarchy effects.

For a single block, in Figure 5a, the memory-only time per intersection is the dominant fraction of the total cost. In addition, it is strongly directionally dependent.

Figure 5b shows results from the same experiment with the data sets blocked into their optimal number of blocks. Appropriate blocking provides improved memory locality, which eliminates many cache misses. This reduces the memory-only time to slightly below the no-memory time for all viewpoints. It also reduces the directional dependence of the memory-only time. Significant further performance gains would require algorithmic changes or a faster processor clock; these could both reduce the no-memory time.

Our next step is to separate the time we attribute to cache miss penalties into L1 and L2 miss portions. Then, we'll compare the total time attributable to cache misses to the memory-only measured time.

Hardware bus-snooping board

The hardware bus-snooping board passively monitors the bus, counting various bus events, including L2 cache misses (L1 misses do not directly generate bus events). Figure 6a shows hardware measurements of L2 misses per intersection for the male data set, divided into several different numbers of blocks. The maximum number of misses per intersection (between frames 70 and 96) is up to 10% higher than 1.0, which means there are slightly more L2 misses than intersections. This is because the bus-snooping board records not only misses from accesses to the data voxels, but also those from all other sources. Blocking into 128 blocks reduces cache misses, but division into 256 blocks does not yield further benefit.

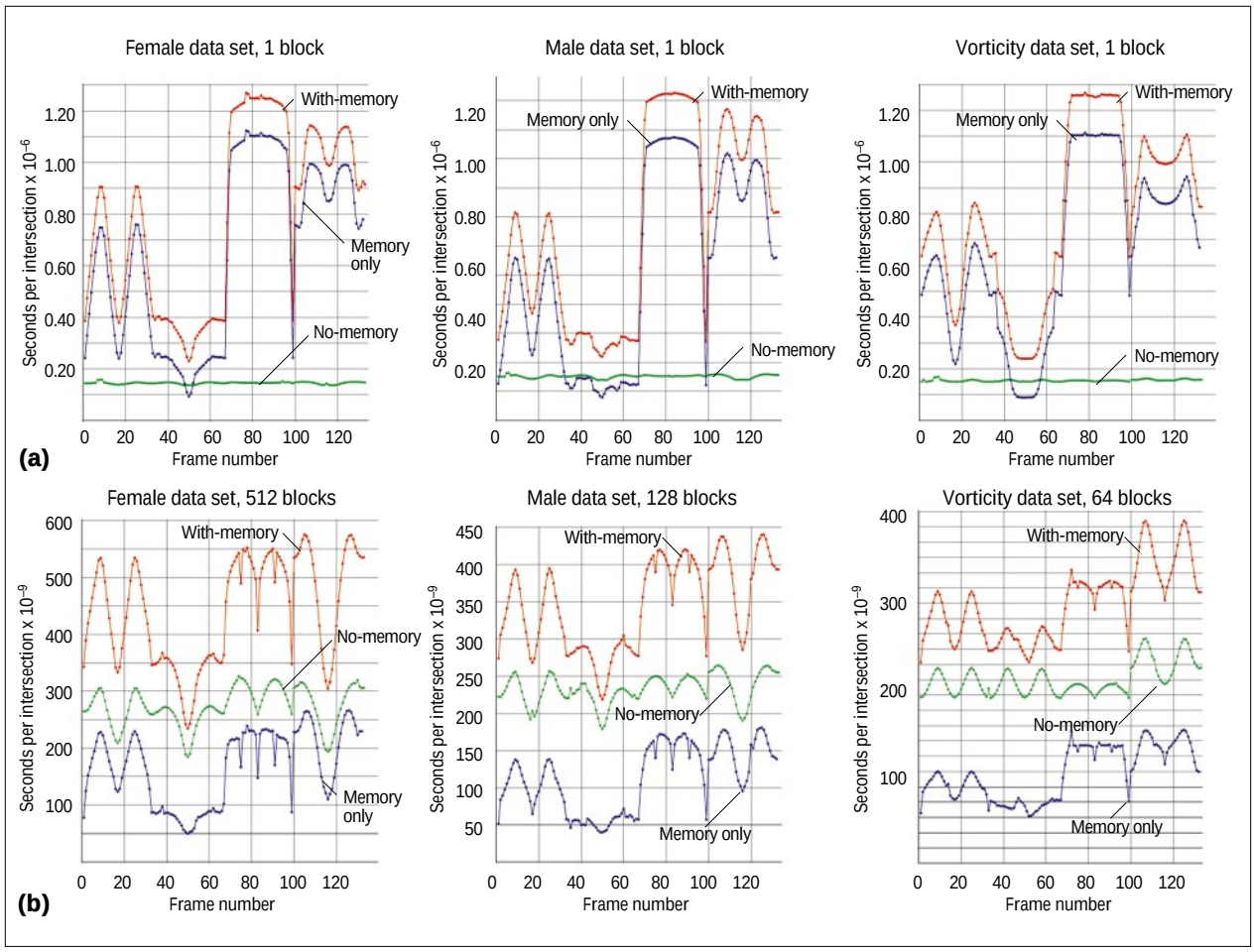

Figure 5. Time per intersection using the with-memory and no-memory algorithms, and memory-only, the difference between the two, (a) for the three data sets each arranged in a single block and (b) each in its optimal number of blocks.

In Figure 6b, for the male data set arranged in one and 128 blocks, we compare the memory-only time to the time we can attribute to L2 misses measured by the bus-snooping board. We derived the latter by multiplying the number of L2 misses per intersection by an L2 miss penalty factor. (The commonly cited L2 miss penalty factor for the R8000 is 690 ns.) The figure shows that L2 misses do not account for all of the memory-only time. (If they did, the red and green lines would match, and the blue and purple lines would match.) Not only is the bus-snoop L2 time too low, it is too low by a nonconstant factor: The 690-ns penalty factor might be low, but that alone could not account for the difference. The memory-hierarchy cost not included here is L1 misses, which we discuss in the next section.

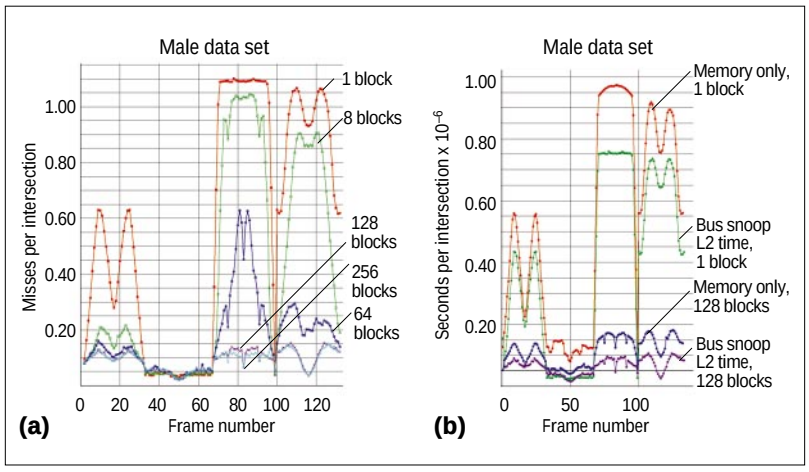

Figure 6. Using the hardware bus-snooping board to analyze memory use: (a) L2 misses per intersection on the male data set; (b) time per intersection as determined using memory-only and the bus-snooping board.

Cache simulator

We built a source-level software cache simulator that replaces the memory read of the current voxel with a call to the simulator. The call returns the proper voxel value, with the side effect of updat-

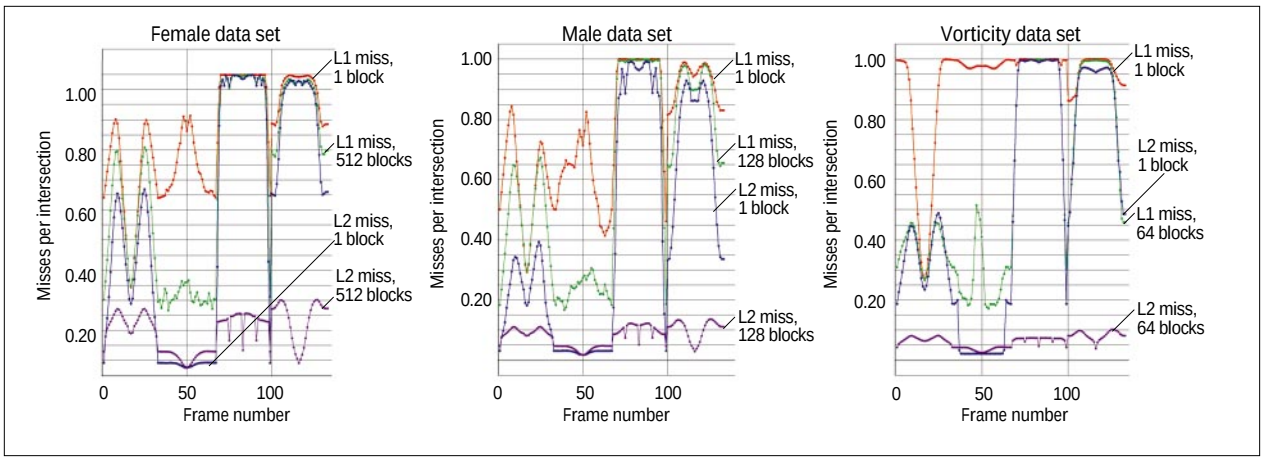

Figure 7. Simulation of L1 and L2 cache misses per intersection for the three data sets.

ing the simulated caches and counting both L1 and L2 cache misses. We use the simulator to count only accesses to the data voxels, which cause the vast majority of all cache misses, ignoring all other memory accesses. We configured the simulator to mimic the caching policies and cache sizes of the R8000: a direct-mapped L1 cache of 32-byte blocks, totaling 16 Kbytes, and a four-way interleaved L2 cache of 128-byte blocks, totaling 4 Mbytes. We used a random block-replacement policy.

Figure 7 shows the simulated counts of L1 and L2 misses per intersection, for the three data sets, for two block sizes each. Here, unlike in the bus-snooping results of Figure 6a, the maximum number of misses per voxel intersection is 1.0, because the simulation counts only cache misses due to accesses to the voxels in the data set.

As we predicted, the L2 miss rate for one block (the blue line) is nearly 1.0 for frames 72 to 94 (when we are looking down the z-axis and adjacent rays do not share an x-z plane), for frames 105 to 127 (when we are looking down a skew axis except for frame 116), and for frame 116 (when we are looking down the y-axis and adjacent rays do not share an x-y plane). Also as we predicted, L1 misses for one block (the red line) are low when we are looking directly down the x-axis (frame 17) but are otherwise relatively high. L1 performance is worse for the vorticity data set than for the other data sets over a wide range of views. This might be because the cubical-vorticity data set fills more of the screen than the other two data sets, so that more rays are at skew angles to the data due to perspective projection.

To analyze more precisely those frames for which we are not looking directly down one of the data set's major axes, we define another set of coordinate axes, shown in Figure A in the sidebar. Whereas the x, y, and z axes are fixed to the voxel data set, we define axes u, v, and n as fixed to the viewer's frame of reference (or alternatively, to the display screen). u points to the viewer's right; v is the viewer's up direction; and n is the direction opposite that in which the viewer is looking. (This makes u, v, n a right-handed coordinate system.)

When a ray accesses a voxel and an L2 cache hit occurs, there are two alternative reasons for that cache line to be already present in the cache:

- It was fetched during a previous voxel intersection by the same ray. We call this a *help-yourself hit*.
- It was fetched by a previous ray. We call this a *help-your-neighbor hit*.

We can expect help-yourself hits to be most common when the n-axis approaches the x-axis—when the viewer is looking directly along x. As n turns gradually away from x, these hits do not drop off sharply, however. With the perspective projection, all rays diverge slightly from n except the one in the exact center of the screen. As n moves away from x, the number of rays that point nearly down x and are still visible on the display screen decreases slowly. Before the point where n becomes perpendicular to x, there are no longer any such rays visible on the screen.

We can expect help-your-neighbor hits to be most common when the u-axis approaches x—when the 128-byte cache lines are aligned with our horizontal scan lines across the screen. In this situation, rays that are cast near one another in time tend to share cache lines. (In fact, the cache lines generated by several rows of rays can be held in the cache at once.)

In Figure 3, throughout frames 34 to 66, u is exactly aligned with x. In Figure 7, this sequence of frames has nearly uniform low L2 miss rates; this is due to many help-your-neighbor hits.

The case of frames 1 to 33 is quite interesting. Frame 1 begins with u aligned with x, so we get many help-your-neighbor hits. As the data set rotates about y, L2 misses increase. By frame 9, we reach a local maximum of L2 misses. However, as we move toward frame 17, where n aligns with x, we begin to get more help-yourself hits. Frame 17 is a local minimum of L2 misses. As the data set continues to rotate, we reach another local maximum of cache misses at frame 25, where x is again 45 degrees between n and u. Miss rates decrease again

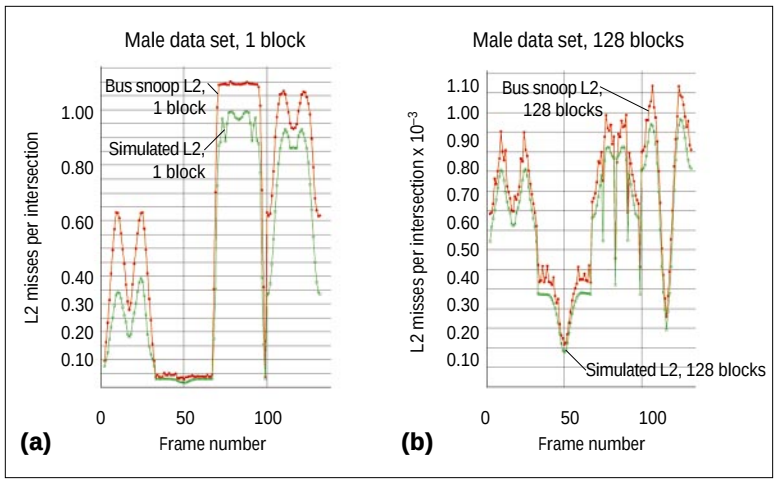

Figure 8. For the male data set in (a) one and (b) 128 blocks, simulated L2 misses per intersection and those detected by the bus-snooping board.

as we approach frame 33, which has many help-your-neighbor hits as u again aligns with x.

In frames 70 through 96, x is aligned with neither n nor u; we expect few help-yourself or help-your-neighbor hits. This region indeed has the worst cache performance. The case is similar for frames 103 to 129, except we expect a small number of help-your-neighbor hits as u gets within 45 degrees of x at frame 116; we observe this effect as a slight reduction in cache misses near this frame.

Figure 8 validates the simulation results, showing the close correspondence between L2 misses per intersection counted by our simulator (green), and those actually measured by the bus-snooping board (red). The simulated misses are always an underestimate of actual misses, because the simulator takes into account only misses arising from access to the current data-set voxel. Not only does this neglect misses due to accesses to other data structures (such as the pixel-accumulation buffer), it also neglects cache thrashing between the data voxels and other data structures. We would expect this error to be greater when L2 miss rates are very high and the cache begins to thrash more heavily. Indeed, note that the pair of lines for 128 blocks (Figure 8b) matches more closely than the pair for one block (Figure 8a). Our simulator yields a slightly low estimate when there are large numbers of cache misses.

This comparison tells us also that the memory access to the current data voxel is the source of the majority of all L2 misses in the algorithm. We cannot directly validate the simulation of L1 misses, because we cannot measure them on the R8000. However, we have some confidence in our simulator because of its success in matching the complex, irregular features of the measured L2 misses.

Figure 9 compares the simulation results for two block sizes against our memory-only time per intersection (red and blue lines). We derived the simulated time per intersection (green and purple lines) from the cache miss counts of Figure 7. We multiplied the counts per intersection of both types of miss by their corresponding penalty factor, and then added them. Commonly cited estimates for L1 and L2 miss penalties are 89 ns and 690 ns.

The resulting simulated time and the corresponding empirical memory-only time lines are strikingly similar in shape for all three data sets and both numbers of blocks. For the optimal number of blocks for each data set, the correlation between the blue and purple lines is extremely close. For one block, the simulated time is consistently somewhat lower than the memory-only time, as explained earlier.

Using the results

We have now precisely identified and characterized the main expense in the ray-casting kernel: L1 and L2 cache miss penalties. We have explained in detail their directional dependence, and demonstrated that they can be controlled by appropriate blocking of the data. Cache miss rates depend primarily on complex interpixel and intrapixel effects, which in turn depend on the relative orientations of the u and n axes (in the viewer's frame of reference) to the x-axis (in the data set's frame of reference). Interblock and interframe cache effects are negligible, because of the high rates at which the cache is completely flushed. We next apply these lessons to parallelization of the algorithm on the shared-memory Power Challenge architecture.

Parallel partitioning and load balance

We examined two combined parallel-partitioning/load-balancing algorithms suitable for ray casting on shared-memory architectures. The first, the *image-tiled* partition, is a 2D static subdivision of the image into M regular rectangles, where M is greater than or equal to N, the number of processors. This algorithm achieves load balance by dynamic self-scheduling. It places tiles on a work queue, and idle processors take tiles from the queue and render the corresponding subrectangle of the entire image. The algorithm sorts the tiles roughly in decreasing order of their cost and achieves good load balance if M is sufficiently large relative to N. We use a tile's cost from the last frame as a predictor for its cost

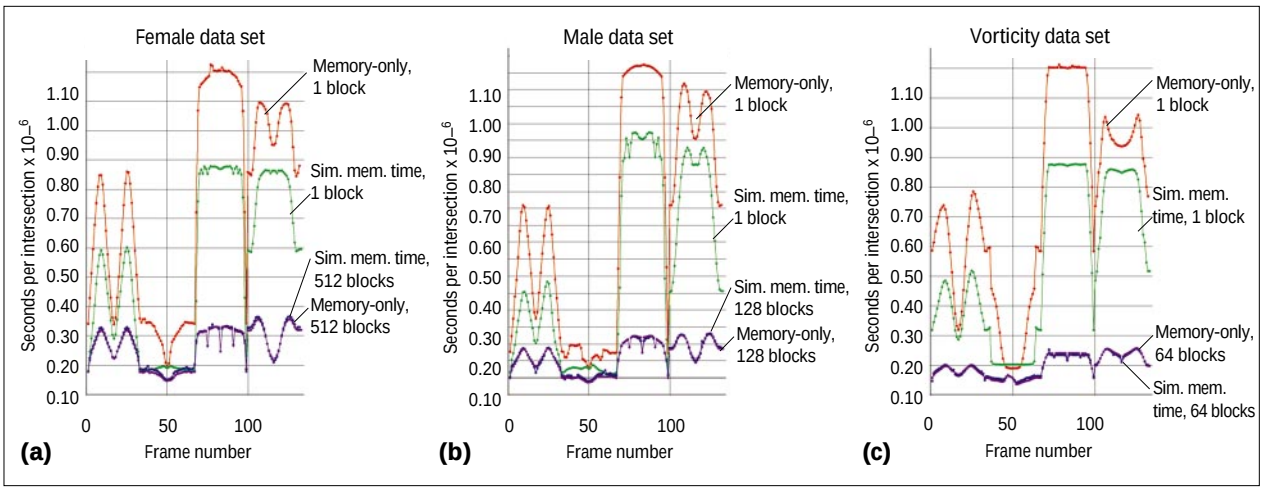

Figure 9. Simulated and memory-only memory hierarchy time per intersection for the three data sets.

in the current frame, exploiting a graphics concept known as viewpoint coherence, the tendency for one viewpoint to be close to the previous one.

The second algorithm, the *object-blocked* partition, divides the 3D data set into M regular rectangular solids, or blocks, with M again larger than N. The algorithm places these blocks on a work queue similar to that used in the image-tiled partition, and load balancing is also similar. Idle processors take an individual block and cast rays through it to generate a tile, the projected image of the block. In its second phase, the algorithm sorts the resulting M tiles in increasing depth order and composites them to generate the entire image. The overhead associated with the compositing phase is proportional to the total area of all tiles. We use an image-partitioning method to parallelize this phase.

The image-tiled partition can easily accommodate global opacity clipping, because a single processor completely determines the disposition of a pixel. In the object-blocked partition, the value of a pixel depends on the contributions of many tiles rendered a priori, so global opacity clipping is infeasible. Intrablock opacity clipping is still possible, but is less effective. Object partitions, however, quite naturally conform to subdivision of the data into blocks that are stored contiguously in memory. This provides the memory-hierarchy performance gains associated with data locality, which we discussed earlier. We can expect the image-tiled partition to have low data-access locality because it treats the data set as one large block.

The Power Challenge shared bus

Multiple processors each sharing the same bus to memory are capable of saturating the bus. To measure the maximum total bus bandwidth of the Power Challenge, we wrote a simple program that causes as many L2 cache misses as possible. It allocates an array many times the size of the L2 cache, and then steps through it at a stride of 128 (missing in L2 each time), reading the values and adding their sum to an accumulator. We ran between 1 and 16 such processes concurrently, measuring a maximum bus bandwidth of 1.0 Gbytes/s. (The machine we used for our experiments had two-way memory interleaving. We also measured the bus bandwidth of an eight-way interleaved machine, and found a maximum bandwidth of about 1.1 Gbytes/s.) Our results showed that more than eight processors making concurrent memory requests as fast as possible saturate the bus. Therefore, other parallel programs with poor cache hit rates might saturate the bus and not exhibit parallel speedup past nine processors.

Experimental results

For the object-blocked partition, we determined empirically that for the female, male, and vorticity data sets, the optimal numbers of blocks were 512, 128, and 64. These numbers are low enough that compositing overhead does not begin to dominate, but high enough to break the data set into blocks smaller than the L2 cache, and high enough to allow good load balance on 16 processors. (We found that this required at least four blocks per processor.) We conducted all of the object-blocked experiments presented here with the optimal number of blocks for each data set.

For the image-tiled partition, we found that there was negligible overhead associated with increasing numbers of tiles, up to at least several thousand. We also found that we needed at least 32 tiles per processor for good load balance from a variety of viewpoints. The optimal number of tiles does not depend on the data set. Therefore, we conducted all image-tiled experiments, for all three data sets, with 512 tiles, yielding 32 tiles per processor for 16 processors.

Parallel speedup

Figure 10a shows parallel speedup of the image partition

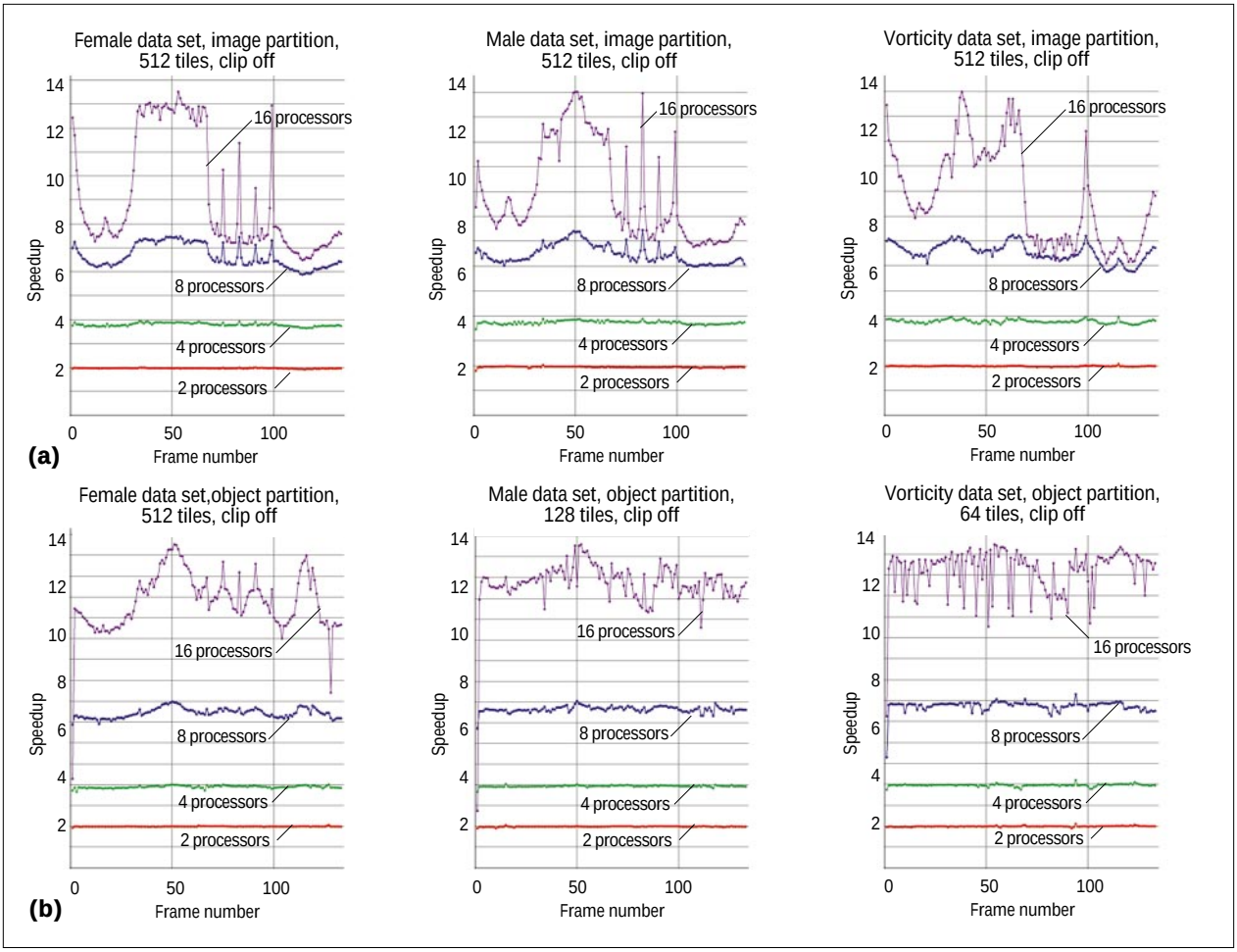

Figure 10. Parallel speedup for multiple processors rendering the three data sets: (a) using the image partition algorithm with opacity clipping off and (b) using the object partition algorithm with clipping off.

on two through sixteen processors compared to a single processor. The image partition does not exhibit parallel speedup past eight processors. The figure shows good parallel speedup on two and four processors for all view directions. However, for views with high L2 cache miss rates, eight processors do not exhibit perfect speedup, and 16 processors exhibit poor speedup. (These views include frames 6 to 12, 22 to 28, 69 to 97, and 102 to 130.) The results match those we obtained with our cache-saturating experiment—that is, the bus becomes saturated by more than eight processors querying it as fast as they can.

As shown in Figure 10b, however, the object partition obtains much better parallel speedup. For all data sets, the speedup is by a factor of more than 7.5 on eight processors. On 16 processors, the speedups average 13, 14, and 14.5 for the female, male, and vorticity data sets. Speedups with opacity clipping turned on are not qualitatively different.

Load balance

Figure 11 shows a measurement of the load balance for the image and object partitions. Our metric of load balance takes the ratio of the average time for all processors to complete a frame to the maximum time for any processor. If the value of this ratio is 1, then the maximum and average are equal, and the time for all processors is equal—a perfect load balance. As the ratio decreases, the average processor spends more time idle waiting for the slowest processor to finish. We used a log scale for these graphs to make small deviations from ideal load balance visible.

For the image partition, in Figure 11a, load balance is extremely good across all data sets and all frames. For all data sets, load balance never falls below .94, and is almost always above .99. The image partition's lack of parallel speedup past eight processors, then, is not due to poor load balance—all processors slow down equally. This accords with our explanation that poor cache hit rates are responsible for the lack of speedup on 16 processors.

For the object partition, Figure 11b, the load balance metric is generally above .95. Only during frames 50 and 116 for the male and female data sets is the load bal-

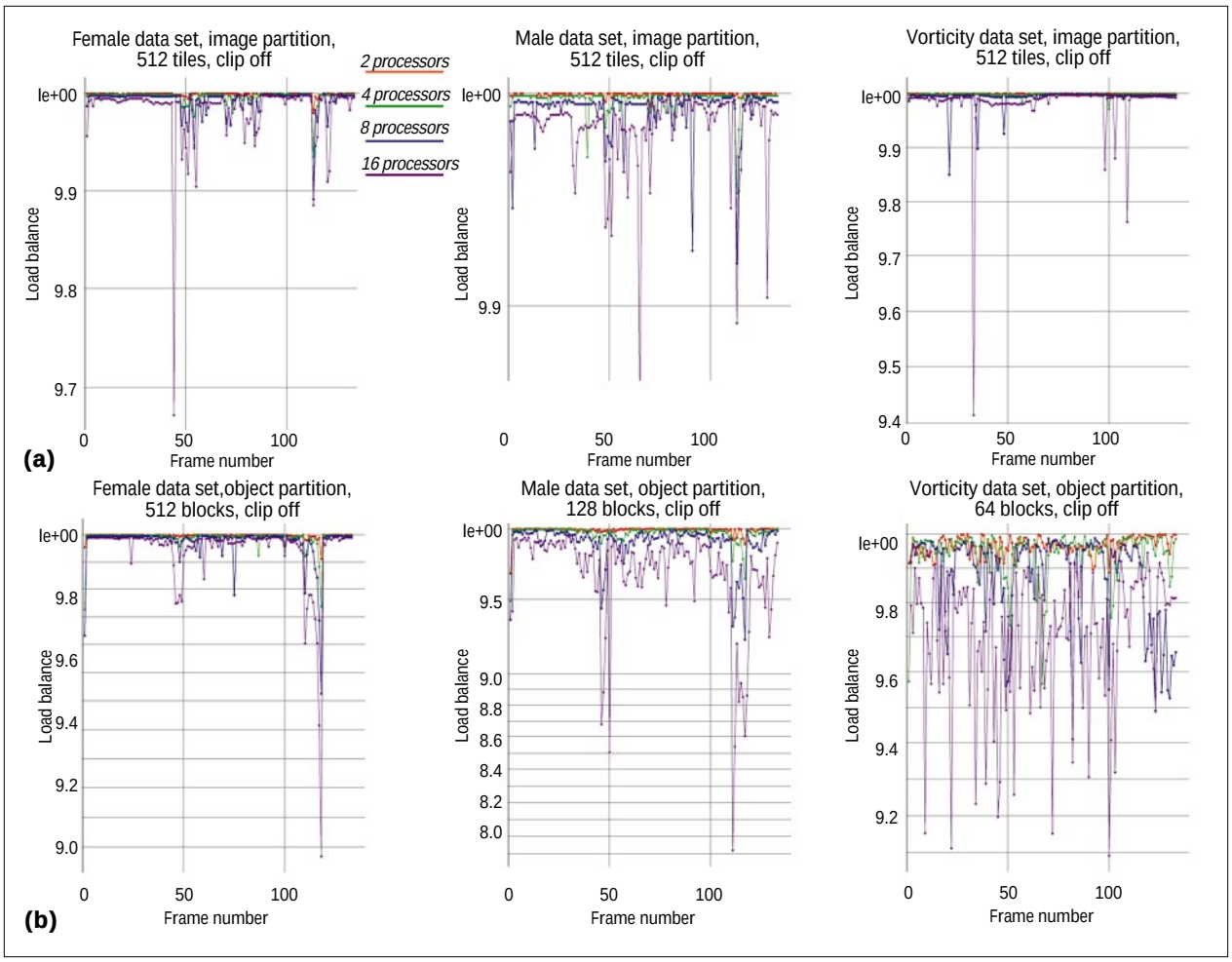

Figure 11. Load balance on multiple processors for (a) image partition and (b) object partition clipping off for both algorithms.

ance slightly worsened. In both of these cases, the feet of the cadaver swing quickly past the viewer. This causes rapid changes in the rendering costs for the blocks in the feet of the cadaver: At one moment they are off the screen, and have no cost; at the next moment they take up a large part of the screen, and have a very high cost. Because we use costs from the previous frame to estimate the cost of the current frame, this rapid change can cause some blocks to be sorted out of order in terms of their true cost, causing load imbalance. Note that the vorticity data set does not show this worsening of load balance at these frames; because the data set is cubical in shape, one "end" of the data set does not sweep particularly close to the viewer at these frames, so the rapid change in costs does not occur.

Comparison of partitioning methods

The image partition is inferior to the object partition as both a serial and a parallel algorithm. On a single processor it simply obtains worse cache performance; on more than eight processors, it saturates the bus and is unable to produce further parallel speedup. However, the image partition does have one advantage over the object partition—it allows global opacity clipping to be effective. The efficacy of this optimization is data-set-dependent, but it can be significant. (The object partition can only apply opacity clipping by the block, which is much less effective.)

Figure 12 compares four variations of the algorithm on eight processors: the image partition with clipping on (red) and clipping off (green); and the object partition with clipping on (blue), and clipping off (purple). The image partition with clipping on is slightly superior to the object partition for the male and female data sets for many of the frames with low L2 miss rates (frames 1 to 61). Both data sets contain a large opaque object, and these algorithmic performance gains outweigh the other factors. However, for frames with worse L2 miss rates, the object partition is superior. Furthermore, on the largely translucent vorticity data set, opacity clipping is less effective. On this data set, the object partition is much superior to the image partition for all views except where L2 miss rates are lowest (frames 42 to 58)—even there, it is slightly superior. Furthermore, the object

partition with clipping off is actually faster than the object partition with clipping on; rays become opaque within a single small block so rarely that it is cheaper not to bother testing for them at all.

To further champion the object partition, Figure 12b compares the same four variations with 16 processors. On 16 processors, the image partition does not run much faster than on eight, but the object partition continues to speed up. For the female and male data sets on 16 processors, the object partition with clipping on is fastest for most views. For the vorticity data set, the object partition with clipping off is again the fastest for nearly all views.

Maximum frame rates

Taking the best partition for each data set in Figure 12b, we calculated the average frame rate across all viewpoints for each data set. For the 1-Gbyte female data set, using the object partition with opacity clipping, on 16 processors at a resolution of 400×300 pixels, we get an average of 1.0 frame per second. This is faster than has been previously cited in the literature for a data set this large. Furthermore, our test suite was designed to include the most expensive view directions in the average. For the 357-Mbyte male data set, using the same partition, we get an average of 1.9 frames per second. For the 128-Mbyte (512^3) vorticity data set, using the object partition without opacity clipping, we get an average of 2.9 frames per second.

We have also extended our methods to a cluster of such machines, replicating the data set on each (admittedly not a scalable solution, but fast). Using eight Power

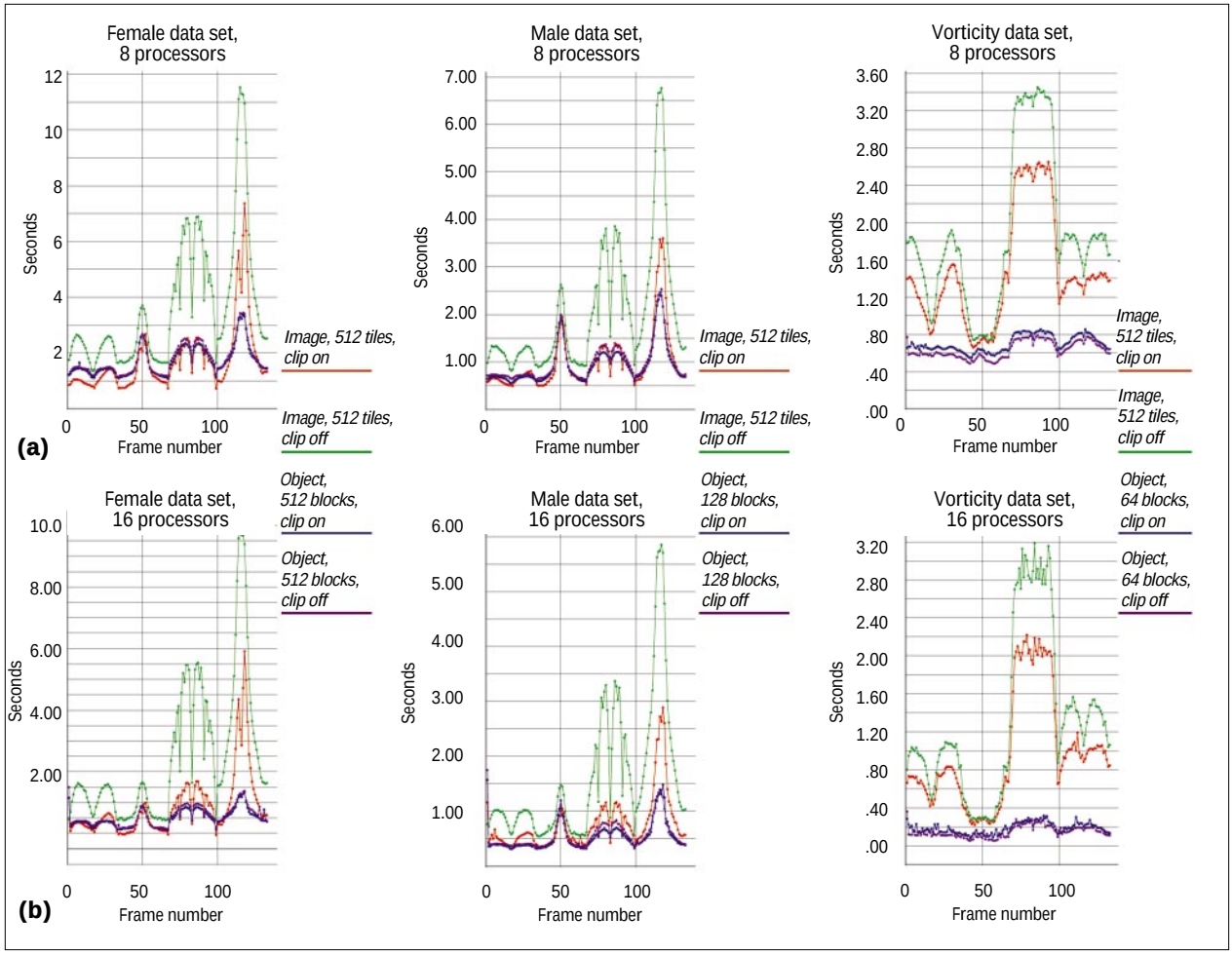

Figure 12. Time per frame of four variations of the algorithm rendering the three data sets, running on (a) eight processors and (b) 16 processors.

Challenge machines with a total of 64 processors, we attain rates up to 10 frames per second on the 357-Mbyte male data set.

Ray casting contains coherence which can be exploited to increase memory locality. The associativity of the composition operator allows us to manipulate the order of memory accesses to increase cache performance, specifically by dividing the data set into appropriately sized blocks. The experimental results in this article and the analytical models they support provide a useful framework to analyze the parallel memory-hierarchy performance of other problems that also contain such coherence. As we have shown for ray casting, good parallel speedup for such problems also relies on the careful optimization of memory-hierarchy performance by exploiting coherence to increase memory locality.

Acknowledgments

We thank the Advanced Systems Division of Silicon Graphics Computer Systems and the National Center for Supercomputing Applications at the University of Illinois, Urbana-Champaign, for their generous provision of compute cycles. Thanks also to H. Ross Harvey of Avalon Computer Systems. Thanks as well to Paul Woodward of the Laboratory for Computational Science and Engineering, University of Minnesota (woodward@lcse.umn.edu) for the vorticity data set. This research was sponsored by the Defense Advanced Research Projects Agency under contract number DABT63-95-C-0116, and Aasert award number N0014-93-1-0843.

References

1. The Visible Human Male and Female data sets, Visible Human Project, c/o Michael J. Ackerman, Nat'l Library of Medicine, http://www.nlm.nih.gov/research/visible/visible_human.html.

2. P. Lacroute, "Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization," *Proc. Parallel Rendering Symp.*, ACM, New York, 1995, pp. 15–22.

3. J.P. Singh, A. Gupta, and M. Levoy, "Parallel Visualization Algorithms: Performance and Architectural Implications," *Computer*, Vol. 27, No. 7, July 1994, pp. 44–55.

4. P. Mackerras and B. Corrie, "Exploiting Data Coherence to Improve Parallel Volume Rendering," *IEEE Parallel and Distributed Technology*, Vol. 2, No. 2, Summer, 1994, pp. 8–16.

5. J. Nieh and M. Levoy, "Volume Rendering on Scalable Shared-Memory MIMD Architectures," *Proc. ACM Siggraph Workshop on Volume Visualization*, ACM, 1992, pp. 17–24.

6. T.W. Crockett, "Parallel Rendering," *Encyclopedia of Computer Science and Technology*, Vol. 43, No. 19, Marcel Dekker, New York, 1996, pp. 335–371; ftp://ftp.icase.edu/pub/techreports/95/95-31.pdf.z.

7. M. Levoy, "Design for a Real-Time High-Quality Volume Rendering Workstation," *Proc. Chapel Hill Workshop on Volume Visualization*, ed. C. Upson, Dept. Computer Science, Univ. of North Carolina, 1989, pp. 85–92.

8. K. Ma et al., "A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering," *Proc. Parallel Rendering Symp.*, ACM, 1993, pp. 15–22.

Michael E. Palmer is a senior engineer in research and development at Inktomi Corporation in San Mateo, California. His research interests include the application of parallel and distributed hardware and algorithms to large problems requiring interactive performance, such as volume rendering and text retrieval. He received his BS in Physics from Yale University, and his PhD in Computer Science from the California Institute of Technology. Direct questions to him via e-mail at mep@inktomi.com.

Brian Totty is the director of engineering at Inktomi Corporation. He has directed the research and development of Inktomi's networked, parallel applications since February 1996. Previously, he was a parallel systems scientist with Silicon Graphics, in Mountain View, California. Totty holds a BS in electrical engineering and computer science from the Massachusetts Institute of Technology. He also holds MS and PhD degrees in computer science from the University of Illinois at Urbana-Champaign, where he received an IBM graduate fellowship and a Darpa fellowship in parallel processing. His e-mail address is bri@inktomi.com.

Stephen Taylor is an associate professor in the Computer Science Department at Syracuse University. His research interests include concurrent computing and multispectral image analysis. He received his PhD from the Department of Applied Mathematics at the Weizmann Institute of Science, Israel. Reach him via e-mail at steve@scp.syr.edu.