Cache-Rings for Memory Efficient Isosurface Construction

David M. Weinstein
Email: dweinste@cs.utah.edu

UUCS-97-016

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

February 22, 1998

Abstract

Processor speeds continue to increase at faster rates than memory speeds. As this performance gap widens, it becomes increasingly important to develop "memory-conscious" algorithms – programs that still optimize instruction count and algorithmic complexity, but that also integrate optimizations for data locality and cache performance. In this paper we present a topological isosurface extraction algorithm which utilizes a "cache-ring" data structure to optimize memory performance. We compare our algorithm to an analogous edge-hashing algorithm which, though functionally equivalent, gives less priority to memory performance. While our algorithm actually executes more instructions during execution, we nonetheless see a speed-up over the traditional method, as we more-than-compensate for the extra instructions with superior memory performance.

1 Introduction

The performance gap continues to grow between applications which haphazardly access data and applications designed to optimize memory access patterns. This trend is a result of the inability of memory speeds to keep pace with processor speeds - while processor speeds double every *two* years, memory speeds double only every *eight* years [1]. In an effort to mitigate this growing differential, memory designers have developed increasingly deep, multi-tiered storage hierarchies. The top tiers (registers and L1 cache) consist of fast storage capable of delivering data to the datapath fast enough that program execution can continue smoothly with little to no delay. The next tiers of storage, deeper (L2) caches and main memory, are not stored as close to the CPU, and as a result accesses to this data can be several orders of magnitude slower. When enough L1 misses accumulate, the CPU generally stalls program execution until the requested data arrives from the memory management unit (MMU). If these stalls occur frequently, as can be the case with haphazard data management, program execution can suffer drastic, even crippling, slow-downs. This condition has only been exacerbated with the advent of parallel architectures. Now, not only must the data be in L1 cache for execution to continue without delay, but it must be in L1 cache on the *right* processor. Improving memory performance can no longer be a matter of hacking the code as an after-thought to improve memory access patterns – memory performance must be treated as a primary factor in algorithm design.

Recognizing this paradigm-shift, we have developed a topological isosurface extraction algorithm to optimize cache performance. Our algorithm exploits the predictability of memory accesses during Marching Cubes [2] surface construction, matching data structures to these access patterns in order to minimize cache misses. The data structure we have developed, termed a *cache-ring*, leverages hardware access mechanisms (such as spatial coherence and cache line fetching) to minimize MMU stalls. Upon analysis, we found that our algorithm actually requires more instructions than the comparable edge-hashing algorithm (most of these increases are due to bookkeeping operations, as we will discuss later); however, we see moderate performance improvements, as our cache-performance more than makes up for this overhead by reducing the number of MMU stalls. As the processor-memory gap widens and parallel processors become increasingly prevalent, we anticipate even greater performance improvements with such memory-conscious algorithms.

2 Background

Isosurface extraction has become a ubiquitous problem in the scientific visualization literature. We are all familiar with Lorensen and Cline's Marching and Dividing Cubes algorithms [2, 3], Wilhelm and Van Gelder's hierarchical algorithms [4], and some of the more recent improvements based on locating the critical points of the field or reparameterizing the domain based on value with active lists [5], span filters [6], both active lists and span filters [7], or span-space [8]. While all of these algorithms fall under the broad umbrella of "isosurfacing" methods, they clearly differ in their domains of applicability. In this paper we will focus on those methods that generate a **connected, topological surface** while extracting. Such a surface is often stored as a list of vertex positions, and a list of triangle triplets (the three vertex indices which compose each triangle). This representation is distinct from the "geometric primitives" format, where each triangle contains its three vertex locations, rather than indices into a vertex list. The geometric representation is useful when the surface's only purpose is to be rendered. In this case, the triangle list can be sent directly to a graphics pipeline for rendering. In contrast, the topological surface is more useful when the surface will be operated on for other purposes such as model editing, boundary condition application, and decimation/refinement. In all of these cases, it is not just important that the surface "look right" but that it be topologically connected.

Isosurface extraction is not the only method used for constructing topological surfaces – indeed, the vision literature is rich with methods for reconstructing models from range data [9, 10, 11]. For the purpose of this paper, we will restrict our interest to topological surface generation from scalar field data. Additionally, while there are many algorithms for extracting isosurfaces from data mapped to unstructured meshes, we will not be discussing those here. For the moment, we restrict our domain of interest to topological surface extraction from scalar field data on regular grids. Such data is prevalent in medical applications, as data from CT and MR scanners is most often comprised of parallel slices (which can then be stacked up to build a regular lattice) or a volume of regular hexahedral elements (voxels).

Within the domain of surface extraction from regularly gridded scalar data, several algorithms have been published. The most intuitive of these, an extension of Marching Cubes, is the seed-growth algorithm [12]. This algorithm was not motivated by the need for topological surfaces (that result was merely a by-product); rather, it was introduced as a way of extracting a single surface connected to a starting node in order to restrict the surface domain. This algorithm's strength also turns out to be a hindrance, though, as the user often wants more than a single connected component as output. Furthermore, this algorithm turns out to be exceedingly difficult to parallelize because the relevant data can not be easily decomposed into discrete regions.

Another topological, regular grid extraction method is the "Bin and Coalesce" method - a generalization of Rossignac and Borrel's algorithm for surface decimation [13]. This algorithm takes the geometric data discussed above and, for each triangle, bins its vertices by location. When every triangle has been processed, vertices located in the same bin are coalesced into a single vertex, and a topologically connected surfaces is output. The drawbacks to this method are that it requires a fair amount of memory overhead, does not necessarily preserve the topology of the initial surface, and must be applied as a post-processing step to an already-extracted surface.

Montani *et al.* implemented a discrete surface construction method that bisects, rather than linearly interpolates, the intersected edges [14]. The advantage to their algorithm is that the output surface can be easily decimated, as the resultant vertices lie within a finite number of indices. However, as with the original Marching Cubes algorithm, they must postprocess their data in order to obtain a connected, topological surface.

Finally, there is the edge-hashing method, as described at the end of Wilhelm and Van Gelder's octree isosurfacing paper [4]. This method uses hashing to create connected surfaces *during* extraction, rather than after. It works from the observation that if linear interpolation is used to locate surface-lattice intersection, then each edge in the lattice contains at most one vertex of the isosurface. Additionally, every isosurface vertex located on an interior (non-boundary) edge of the volume, will be found by exactly three other voxels - namely those voxels which share that edge. Working from these observations, Wilhelm and Van Gelder add each vertex location to their list the first time they encounter it, and then store the new vertex index in the hash table based on its edge. When this index is needed by the adjoining cells in the future, it is retrieved from the hash table. Finally, in order to conserve memory, they remove each interior vertex from the hash table after the fourth (and, by definition, final) time it is accessed. This method is very efficient, and succeeds in building a topologically correct surface as output. Furthermore, it can be parallelized in a straightforward manner - the only complication coming from sharing information about vertices on the boundaries between processors.

While these algorithms capitalize on much of the continuity of the data, what they fail to exploit is the *predictability* with which the data is accessed. In short, none of the above algorithms has very good cache performance, and hashing, in particular, has notoriously bad cache performance. As memory performance becomes an increasingly important factor in overall algorithm performance, it becomes necessary to give high priority to memory performance when designing algorithms. This is precisely what we have done in implementing our new topological isosurface extraction algorithm: recognizing the possible structure of memory accesses for topological isosurface construction, we have built a data structure and algorithm to match these access patterns.

3 Methods

We chose a Marching Cubes style algorithm as our base algorithm, opting for its "embarrassingly parallel" structure over the more elegant, but somewhat irregular, structure of a seed-growth method. The algorithm consists of moving linearly through the data, row by row, slice by slice, examining each cell for a possible intersection with the isosurface. When an intersected cell is located, the edge intersections are computed and triangles are generated to span those intersections. Viewed all together, these triangles represent a geometric piecewise linear set of surfaces where each surface either intersects the boundary of the volume or is closed.

3.1 Predictable Data Traversal

As we march through the data and generate these triangles, we recognize that every interior edge intersection (*i.e.* any edge that is not on a bounding face of the volume) is shared by exactly four voxels. If an edge is intersected by a triangle in one voxel, it will necessarily be intersected at exactly the same point by triangles in the other three voxels as well (because Marching Cube surfaces are manifold and piecewise continuous). Without loss of generality, let us assume that we are traversing our data in x-major order (*i.e.* the data is stored as consecutive slices of x, with z-coordinates changing fastest). If we are currently examining voxel $v[i, j, k]$, and we find an intersection point along its $(+x, +y)$ edge, then we know that when we reach voxel $v[i, j+1, k]$ we are guaranteed to find a triangle with this same intersection. Similarly, this intersection point will be repeatedly found in voxel $v[i+1, j, k]$ and voxel $v[i+1, j+1, k]$. Since we can predict exactly when we will be arriving at voxel $v[i+1, j, k]$, we should be able to "prefetch" all of the precomputed edge intersections so we can reuse their values "on the fly." If the size of our volume is $(nx * ny * nz)$ nodes, then we have $((nx-1)*(ny-1)*(nz-1))$ voxels. It follows that we will arrive at voxel $v[i, j+1, k]$ exactly $(nz-1)$ iterations later; at voxel $v[i+1, j, k]$, $((nz-1)*(ny-1))$ iterations later; and at voxel $v[i+1, j+1, k]$, $((nz-1)*ny)$ iterations later.

3.2 Cache-Ring Data Structure

This brings us to the notion of cache-rings. For any given voxel, $v[i, j, k]$, in order to generate the portion of the surface that intersects that cell, we only need the scalar field data at the corners of the cell, and the shared-edge intersection information from the previously visited voxels: $v[i, j, k - 1]$, $v[i, j - 1, k]$, $v[i, j - 1, k - 1]$, $v[i - 1, j, k]$, $v[i - 1, j, k - 1]$, and $v[i - 1, j - 1, k]$. These voxels and edges are shown in Figure 1.

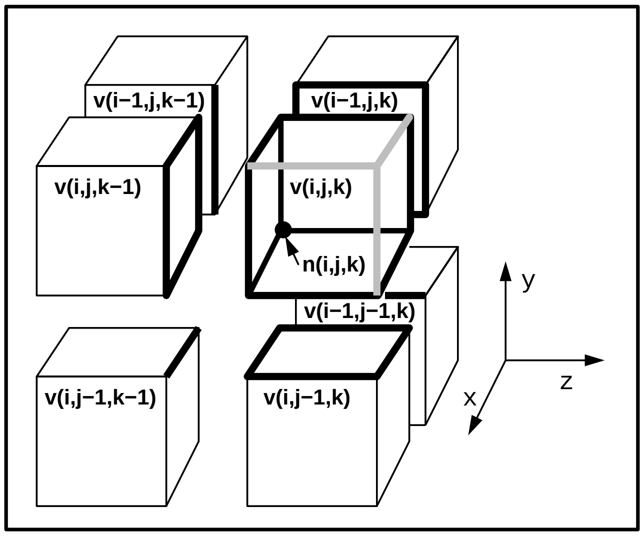

Figure 1: Previously visited, neighboring voxels. The thick, dark lines indicate edges shared with the current voxel, $v[i, j, k]$. The location of the node $n[i, j, k]$ is indicated with a dark point. The lighter gray lines on the current voxel indicate edges which have not yet been encountered, and whose intersection nodes might, therefore, need to be computed. Thinner lines indicate all other voxel edges - none of these are of interest when extracting at the current voxel.

The edge information from each direction is stored in a data-structure called a "cache-ring." Each ring is implemented as a ring-buffer, and stores edge intersection information for the neighboring cells of a particular direction. We note that each ring has a maximum length equal to the the number of cells traversed between that neighboring voxel and the current voxel. For example, voxel $v[i, j - 1, k]$ is $(nz - 1)$ voxels away from the current voxel $v[i, j, k]$; therefore, the ring in the y-direction, the y-ring, needs to have a maximum of nz entries (both sides of the current voxel need to be stored, which increases the total by one entry). We traverse the data in normal Marching Cubes fashion, and upon completing each voxel, we record the current information into all of the rings and advance the pointers. A graphical depiction of three of these rings is shown in Figure 2.

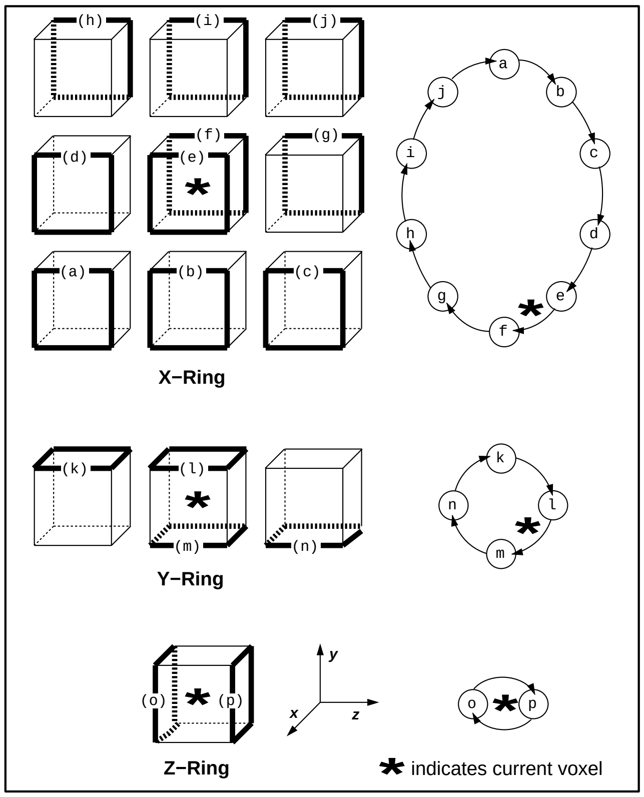

Figure 2: The X-, Y-, and Z-Rings and their contents during data traversal. The left side of each ring shows the currently active edges whose data is stored in the ring. As indicated by the axes, the cells are visited from left to right, from top to bottom, and from back to front. The right side of each diagram depicts the cache-ring storing the edge information, with arrows indicating the direction of traversal.

In practice, we actually utilize only a small portion of each ring at a time for storing precomputed indices - most of the voxels between the current voxel and the last voxel in the ring contain no intersection points, and are empty. Recognizing that ring data is always accessed deterministically (every entry which is stored is guaranteed to be accessed, and the accesses will occur in exactly the same order in which they are stored), we only need to store those edges which contain an intersection. With this revised implementation, we now store two indices into each ring - one for the voxel being examined and one for the neighbor information being stored (previously, these indices were identical, so a single value was sufficient). New data is always inserted at the last index, and the next intersection index is always read from the head.

3.3 Algorithmic Performance

Implementationally, the cache-ring algorithm operates very similarly to the hash table - the primary difference is in the data structures used. It turns out that even with a very fast hashing function, the hash table method is slowed because the table entry for that edge is rarely still in fast cache when it is accessed by subsequent voxels. And, most-importantly, **when we suffer a cache miss for an edge, we are just as likely to suffer a miss on lookups of subsequent edges in the same direction.** This is not the case with the cache-ring algorithm. This is because on a cache miss the memory management unit (MMU) brings in in entire cache line of data, rather than just the single requested item. This standard technique is generally efficient because of the law of data locality: *if a memory address is being accessed now, its neighbors are likely to be accessed in the very near future* [1]. We turn this mechanism to our advantage with cache-rings, as we essentially prefetch most of our data.

In practice, we did not see as much speed-up as we had anticipated. This turned out to be because we were knocking our own data out of the cache as we tried to keep six rings stored there at the same time. Realizing that all of the necessary data is redundantly stored in multiple rings, we can begin carefully merging some of the rings.

The first improvement comes from recognizing that ring $r[i, j-1, k]$ is exactly the same length as ring $r[i, j-1, k-1]$, and the data is all the same – just shifted over one. To avoid colliding with ourselves in the cache, we combine these lists by zippering them together into a single ring. The same process is applied with rings $r[i-1, j, k]$ and $r[i-1, j, k-1]$. The second improvement comes from removing $r[i, j, k-1]$, since this ring only has one element! Instead, the current and next values from this ring are stored in global variables.

This leave us with three cache-rings: $r[i, j-1, k]$, $r[i-1, j, k]$ and $r[i-1, j-1, k]$. Our final optimization is to eliminate the last ring, $r[i-1, j-1, k]$ by combining it with $r[i-1, j, k]$. This operation introduces a complication, though, because ring $r[i-1, j-1, k]$ is nz voxels longer than ring $r[i, j-1, k]$. We clean this up by initially storing the $r[i-1, j-1, k]$ data in ring $r[i, j-1, k]$ and then copying it into $r[i-1, j, k]$ when the first ring has cycled. For our final implementation, we are left with two cache-rings then: $r[i-1, j, k]$ and $r[i, j-1, k]$. For simplicity, we will refer to these as **RX** and **RY**, respectively.

To visualize how these data structures serve the algorithm, we will follow through a short example. If we are extracting an isosurface of value v and some node $n[i, j, k]$

(see Figure 1) has value $(v + .1)$ and node $n[i + 1, j, k]$ has value $(v - .1)$, then the triangular, linearly interpolated isosurface S, which will be extracted from the volume, will contain the point p, located at the midpoint of the edge between node $n[i, j, k]$ and node $n[i + 1, j, k]$. This edge is shared by four voxels. Specifically:

- the $(+y, +z)$ edge of voxel $v[i - 1, j - 1, k]$ and
- the $(+y, -z)$ edge of voxel $v[i - 1, j, k]$ and
- the $(-y, +z)$ edge of voxel $v[i, j - 1, k]$ and
- the $(-y, -z)$ edge of voxel $v[i, j, k]$.

This intersection point is first encountered from voxel $v[i - 1, j - 1, k]$. Its location along the edge is computed, and the index to this new point is placed in rings **RX** and **RY**. The algorithm then continues processing cells. When it arrives at voxel $v[i - 1, j, k]$, it finds that it has an intersection on the $(-y, +z)$ edge and blindly grabs the index of point p from cache-ring **RY**, knowing that the leading entry must be this intersection node's index. After all the triangles have been generated for voxel $v[i - 1, j, k]$, this intersection point is again inserted into the **RX** cache, so that the point can be referenced in voxel $v[i, j, k]$.

The only specials cases in this implementation come when we encounter non-interior (exterior) edges. As defined above, these are edges which lie on boundary faces of the volume. When we are examining such a cell, we need to take special care to not assume the usual edge intersections have been precomputed and cached.

3.4 Multiprocessor Implementation

For the multiprocessor implementation of the cache-rings algorithm, we divide the dataset along the x-dimension into "slabs," which are independently handled by separate processors. Each processor stores its first **RX** cache-ring globally, so the data can be accessed by the processor sharing those voxel edges. All subsequent cache-ring loads and stores are done on a local cache-ring, until the final plane of voxels in the slab is reached. At this point, the processor waits until the processor sharing the boundary edges of those voxels indicates that it has finished processing storing that edge information in its global **RX** cache-ring. In practice, this synchronization doesn't cause any slowdown, because every processor has generally completed processing its first plane of voxels before any processor is ready to process its last plane.

The dimension of the slabs and the number of utilized processors can be selected by the user. We found that choosing slabs with less than four voxels of thickness lead to dramatic slowdowns, as increasing percentages of execution time became necessary for sharing and copying data at the boundaries.

When each processor has completed its portion of the surface, the pieces are serially evaluated by a single processor to determine the offsets of the indices in each partial surface. These offsets are then read in by the processors and, once again proceeding in parallel, each processor adjusts the indices of the triangles it has extracted. Once the indices are consistent, the points and triangles can be moved directly into a single data structure which stores the entire surface.

Performance analysis shows that there is not much delay incurred by the barrier synchronizations described above. The largest performance hit comes from the extra work needed to process boundary voxels. When the entire volume is a single slab (as is the case for single processor execution), there are a minimal number of voxels touching boundaries. With each added processor, the size of the slabs decreases, and the percentage of the voxels which share a boundary rapidly rises. Furthermore, processing voxels which touch the Y and Z boundaries of the model incur a smaller overhead than voxels which are at an X boundary, and it is the number of X boundary voxels which is increasing as we divide the volume into more, ever thinner slabs.

3.5 Edge Hashing

For comparison, we also implemented an edge-hashing extraction/construction method that builds the same connected surfaces as out cache-ring method. Here again we exploit the predictability of edge accesses, recognizing that an interior node will be accessed by exactly four voxels. After a node has been accessed four times, we delete it from the hash table, thus saving time (less likely to have collisions during future lookups) and memory (only the active node indices are stored).

Every edge spans two voxels. For the purpose of defining a unique, consistent hash-key for each edge, we use the "smaller" node as the root of our index. (By "smaller" we mean the node to the left, below, or behind the other node – see Figure 1 for reference.) The i, j, and k indices of this node are concatenated to form the high-order bits of the key. These are followed by two bits to encode direction information (right, up, or forward), and two final bits to encode how many times the node has been referenced. If we have a data set containing less than 2^{28} (134 million) voxels, then this hash-key can be constructed as a 32-bit integer. Alternatively, for larger data sets, we can hash long integer keys.

The multiprocessor implementation of the edge-hashing method utilized the same synchronization mechanisms as the cache-ring version. The only difference is that a separate hash table is created for the boundary voxels. This global object is analogous to the shared boundary cache-ring discussed above.

4 Results

We implemented three algorithms for timing comparisons: standard Marching Cubes (only building unconnected triangles), edge-hashing, and cache-rings. These algorithms were all timed for isosurface extraction on four radiology data sets. The first data set is a low-resolution, $(32 * 32 * 32)$, MRI model of a head. The second data set is the same model, but at twice the resolution in each dimension, $(64 * 64 * 64)$. The third data set is the same individual's head imaged during a different MRI scan and contains $(56 * 512 * 512)$ nodes. The final model is the CT thorax of the Visible Human Data Set$^{(TM)}$ [15] and contains $(512 * 512 * 168)$ nodes.

For the MRI data sets, we extract an isosurface values corresponding roughly to the scalp. The CT data was isosurfaced at a value more appropriate for bone. Screen images of the extracted surfaces from all of these data sets are included at the end of the paper (Figures 5-8).

All three algorithms were run on an SGI Power Onyx, with 14 90 MHz R8000 CPU's. This machine has a shared memory architecture, with 128 byte cache line, 16 Kb of L1 cache, 4 Mb of L2 cache, and 4 Gb of main memory. An L1 cache miss costs, on average, 5 cycles, and an L2 cache miss costs, on average, 100 cycles. For different timings, we utilized different numbers of the available processors. We note that the Power Onyx architecture has a main bus which can be saturated by as few as five processors during heavy utilization.

Model	Time* (sec)	Marching Cubes	Edge Hashing	Cache Rings
32x32x32 Head	EXT	0.47	0.63	0.61
	OH	N/A	0.01	0.01
64x64x64 Head	EXT	2.82	3.60	3.54
	OH	N/A	0.09	0.08

* key:		
	EXT	– Extract Timing
	OH	– Overhead Timing

Figure 3: **Table 1** Timings for single processor isosurface extraction of two data sets, using Marching Cubes, edge-hashing, and cache-ring algorithms.

4.1 Single Processor Timings

Table 1 contains timing results from running the three isosurfacing methods on the smaller MRI data sets, using only a single CPU. The edge-hash and cache-ring build both a geometric and a topological surface, and so we have included timings for the "Overhead" of allocating and copying the topological surface. This permits a more consistent comparison between these methods and the Marching Cubes methods.

4.2 Multiprocessor Timings

Table 2 contains timing results from running the three isosurfacing methods on the three larger data sets and using varying numbers of processors. Once again we have included separate timing values indicating the amount of overhead in the hash table and cache-ring runs. All timings have been summed over the utilized processors. Additionally, we have included a separate "Wall Clock" timing, to demonstrate the amount of real-world time that passed during execution.

4.3 Analysis

From the tables above, it is clear that the basic Marching Cubes algorithm is considerably faster than the algorithms which construct topologically connected surfaces. When connected surfaces are required, edge-hashing and cache-ring extraction methods produce comparable timing results. This second point is somewhat surprising, given the increased book-keeping coding necessary for maintaining the rings. We

Model	CPUs	Time* (sec)	Marching Cubes	Edge Hashing	Cache Rings
64 x 64 x 64 Head (250 K voxels, 47 K triangles)	2	EXT	2.84	4.13	4.07
		OH	N/A	0.12	0.10
		WC	1.54	2.29	2.27
	4	EXT	2.85	5.35	5.30
		OH	N/A	0.15	0.15
		WC	0.84	1.63	1.61
	8	EXT	2.90	7.65	7.80
		OH	N/A	0.35	0.33
		WC	0.50	1.29	1.26
	14	EXT	3.19	12.83	12.70
		OH	N/A	0.51	0.47
		WC	0.40	1.25	1.22
56x512x 512 Head (14 M voxels, 2.4 M triangles)	8	EXT	156.34	401.52	385.21
		OH	N/A	13.21	12.42
		WC	29.65	71.50	68.91
	14	EXT	158.09	613.74	599.72
		OH	N/A	12.46	12.01
		WC	19.28	59.29	58.03
512x512x 168 Torso (44 M voxels, 652 K triangles)	8	EXT	246.70	263.94	251.52
		OH	N/A	2.13	2.09
		WC	29.65	44.42	40.07
	14	EXT	246.88	252.77	240.38
		OH	N/A	3.28	3.06
		WC	22.94	28.63	25.19

* key:		
	EXT	– Extract Timing
	OH	– Overhead Timing
	WC	– Wall Clock Timing

Figure 4: **Table 2** Timings for multiprocessor isosurface extraction of three data sets, using Marching Cubes, edge-hashing and cache-ring algorithms, and varying numbers of CPU's.

conclude that the lack of performance degradation (and actually a small performance improvement) is due to improved cache performance.

In analyzing the performance of the parallel methods, we see that speedups are sub-optimal for all of the algorithms. Analyzing the Extract timings, we see that Marching Cubes had the best speedup, with very little extra computation required when the work was spread across processors. As expected, edge-hashing and cache-rings had added computation when split across processors. This increase reflects the overhead of sharing data at boundaries during triangle extraction, and merging the boundary data during surface construction.

Examining the Wall Clock timings and total Extraction timings from the various methods, it is clear that we didn't have exceptional load-balancing performance. This is an artifact of the uneven distribution of data values, as would be expected from radiological patient images.

A final point of interest is that these performance numbers come from execution on 90 MHz R8000 processors. These MIPS processors are relatively lenient in the penalties they impose for an L1 cache miss. This is primarily due to the fact that they are running at a relatively slow clock rate. The more stringent cache miss penalties imposed on faster processors will further improve the relative performance of cache-ring methods over hashing methods.

5 Conclusions

We have presented a new data structure, the cache-ring, and an augmented Marching Cubes algorithm for the construction of topologically correct surfaces. The details of the algorithm and the data structure implementation are not the points of relevance here. Rather, the primary reason for this paper has been to emphasize the importance of memory-conscious algorithms. By re-engineering our program so it maps to the underlying hardware, we have partially alleviated the bottleneck of accessing memory during isosurface construction.

As we continue to develop new and improved isosurface extraction methods, we believe it is important to keep abreast of the simultaneous shifts evolving in computational hardware development. The performance bottleneck in modern architectures has shifted from computation to memory accesses. This bottleneck becomes even more pronounced in the case of parallel architectures, where data can be distributed among multiple CPU's. As the bottleneck shifts, algorithm development must shift as well, if methods are to remain optimal in practice. The cache-ring data structure is an example of updating a method to follow the underlying hardware's bottleneck shift. Rather than suffer performance degradations with the random data accesses of existent codes, we have developed an augmented algorithm and data structure to alleviate memory-induced stalls.

6 Future Work

Having optimized the surface construction method for regularly-gridded scalar fields, we will now being looking at memory access patterns for other commonly-used algorithms, such as isosurface extraction on irregular grids, and streamline advection.

We look forward to running our code on MIPS R10000 processors in the near future to obtain accurate statistics on cache performance. We would also like to obtain performance statistics on machines with other memory architecture configurations for comparison. Additionally, we would like to have accurate counts of the number of instructions executed for the various surface extraction methods.

In the mean time, we will investigate developing a small simulator to derive cache-performance and instruction count data for those architectures where such statistics are not available. As we can better micro-profile our code to determine architecture-dependent slow-downs, we will continue to develop new algorithms and data-structures which better map to the underlying architectures.

Finally, we have not addressed any of the load-balancing issues which arise from multiprocessor implementations. It would be reasonable to vary the slab thickness in order to accommodate the varying number of surface intersections throughout the volume. We will investigate such load-balancing issues in future versions of this algorithm.

7 Acknowledgments

This work was supported in part by the National Science Foundation. I would like to thank Peter-Pike Sloan, Helen Hu, Steve Parker, Chuck Hansen and Chris Johnson for their valuable comments and suggestions.

References

[1] D.A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2nd edition, 1996.

[2] W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, 1987.

[3] H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford, and B. C. Teeter. Two algorithms for the three-dimensional construction of tomograms. *Medical Physics*, 15(3):320–327, 1988.

[4] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, 1992.

[5] M. Giles and R. Haimes. Advnaced interactive visualization for cfd. *Computing Systems in Engineering*, 1(1):51–62, 1990.

[6] R.S. Gallagher. Span filter: An optimization scheme for volume visualization of large finite element models. In *Visualization '91*, pages 68–75. IEEE Press, 1991.

[7] H.W. Shen and C.R. Johnson. Sweeping simplices: A fast isosurface extraction algorithm for unstructured grids. In *Visualization '95*, pages 143–150. IEEE Press, 1995.

[8] H.W. Shen, C.D. Hansen, Y. Livnat, and C.R. Johnson. Isosurfacing in span space with utmost efficiency (issue). In *Visualization '96*, pages 287–294. IEEE Press, 1996.

[9] B. Curless and M. Levoy. A volumetric method for building complex models from range data. In *ACM SIGGRAPH Computer Graphics*, pages 303–312. IEEE Press, 1996.

[10] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. In *ACM SIGGRAPH Computer Graphics*, pages 71–78. IEEE Press, 1992.

[11] G. Turk and M. Levoy. Zippered polygon meshes from range images. In *ACM SIGGRAPH Computer Graphics*, pages 311–318. IEEE Press, 1994.

[12] R. Shekhar, E. Fayyad, R. Yagel, and J.F. Cornhill. Octree-based decimation of marching cubes surfaces. In *Visualization '96*, pages 335–342. IEEE Press, 1996.

[13] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering complex scenes. In B. Falcidieno and T. Kunii, editors, *Modeling in Computer Graphics: Methods and Applications*, pages 455–465. Springer-Verlag, 1993.

[14] C. Montani, R. Scateni, and R. Scopigno. Discretized marching cubes. In *Visualization '94*, pages 281–287. IEEE Press, 1994.

[15] National Library of Medicine. The visible human project, 1995.

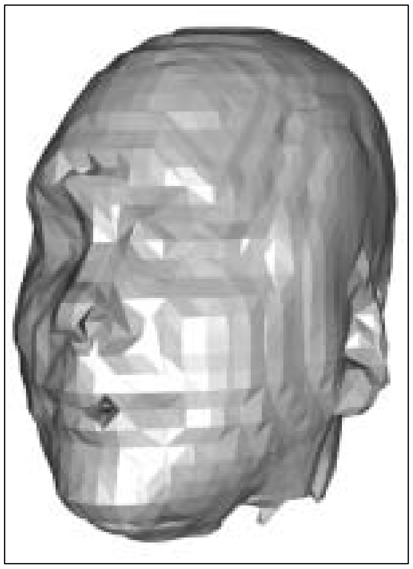

Figure 5: Scalp isosurface from 32*32*32 (32K) node MRI model of a head. Surface contains 9492 triangles.

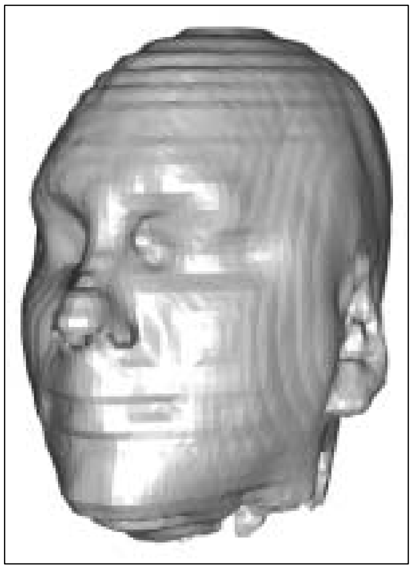

Figure 6: Scalp isosurface from 64*64*64 (262K) node MRI model of a head. Surface contains 47,436 triangles.

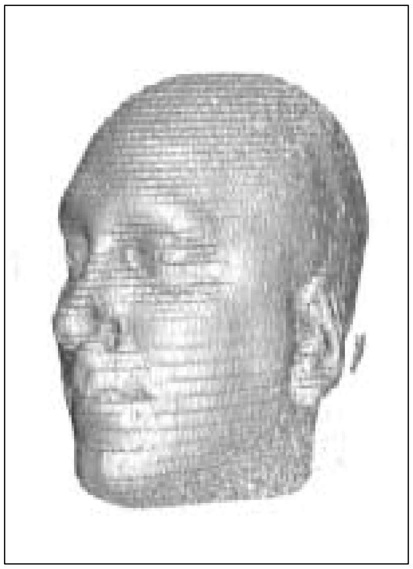

Figure 7: Scalp isosurface from 56*512*512 (14M) node MRI model of a head. Surface contains 2,399,382 triangles.

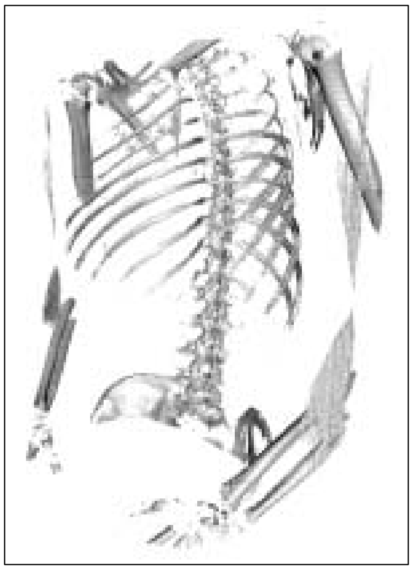

Figure 8: Skeletal isosurface from 512*512*168 (44M) node CT model of Visible Human Project male thorax. Surface contains 651,594 triangles.