Dynamic Ray Scheduling for Improved System Performance

Paul Arthur Navrátil, Donald S. Fussell and Calvin Lin*
Department of Computer Sciences
The University of Texas at Austin

Abstract

The performance of full-featured ray tracers has historically been limited by the hardware's floating point computational power. However, next generation multi-threaded multi-core architectures promise to provide sufficient CPU power to support real time frame rates. In such systems, the emerging problem will be limited memory system performance in terms of both on-chip cache and DRAM-to-cache bandwidth. This paper presents a novel ray tracing algorithm that significantly improves both cache utilization and DRAM-to-cache bandwidth. The key insight is to view ray traversal as a scheduling problem, which allows our algorithm to match ray traversal computations and intersection computations with available system resources. Using a detailed simulator, we show that our algorithm reduces the amount of geometry brought into the cache by up to 32× for primary rays and up to 60× for shadow rays, in exchange for the small overhead of maintaining the ray schedule. Moreover, our algorithm creates units of work that are more amenable to parallelization than traditional Whitted-style ray tracers.

Index Terms: I.3.7 [Computer Graphics]: Ray Tracing—

1 Introduction

Full-featured ray tracing can produce high-quality images but not yet at interactive frame rates. Floating-point CPU power has traditionally been the limiting factor, but modern CPUs have partially removed this barrier. Several current systems trace primary and hard-shadow rays, generated from point lights, at interactive rates[SWW*04, RSH05, WSS05, WIK*06]. New chips with many processing cores promise to overcome this computational bound on ray tracing. Moreover, ray tracing's embarrassingly parallel nature seems to lend itself well to such architectures. However, these multi-core architectures introduce a new bottleneck in the memory-system, because cache and bandwidth must be shared among many cores. This contention is exacerbated by the use of a complex lighting model, which is necessary for photo-realistic images. Complex lighting algorithms can generate incoherent memory accesses (e.g., Monte-Carlo methods, photon mapping[Jen96]) and can require the use of additional data structures (e.g., photon mapping[Jen96]). We conclude, then, that real-time ray tracing of dynamic scenes with complex lighting could be feasible if the ray tracing algorithms could be made memory efficient.

Recursive ray tracers, derived from Whitted's algorithm[Whi80], are not memory-efficient because they traverse rays depth-first. Consecutive primary rays may be tested for intersection against the same geometry, but these tests can be widely separated in time. For example, all child rays of the first primary ray must be traversed before the second primary ray can begin. If the scene is small enough or the cache large enough, the impact of this inefficiency may be masked, but the trend is toward larger scenes rendered using a complex lighting model. Optimizations such as tracing rays in SIMD-friendly packets[WSBW01] or using ray frustums[RSH05] help, but only if rays are sufficiently coherent, which is typically only the case for primary and perhaps shadow rays. These techniques can be considered simple scheduling schemes designed to improve the memory access behavior of the ray tracer. Unfortunately, since they are designed specifically to work for already coherent sets of rays, they show little promise to be of much benefit in handling the incoherent rays in a globally illuminated scene.

Pharr, et al.[PKGH97] use a somewhat more sophisticated scheduler to improve the memory efficiency of ray tracing scenes much too large to fit into main memory. They schedule rays for processing according to their location in scene space independently of their spawn order. The rays traverse the cells of a uniform grid, and they are queued at any cell that contains geometry. When a cell is selected for processing, all rays queued at that cell are tested for intersection against geometry in the cell. Any rays that do not intersect an object traverse to the next non-empty cell. This approach significantly reduces bandwidth usage between disk and main memory and increases the utilization of geometry data in main memory. However, the algorithm is not suited for managing traffic between main memory and processor cache because it allows ray state to grow unchecked, and because the acceleration structure does not adapt to the local geometric density of the scene. These two factors create work loads of highly variable sizes, the effects of which are masked at main memory scale (hundreds of MB) but cannot be masked at cache scale (hundreds of KB).

In this paper, we present an algorithm that schedules ray processing by *actively managing* both ray and geometry state to maximize cache utilization and bandwidth utilization without exceeding peak bandwidth supply. We view the algorithm of Pharr, et al., and Whitted's recursive ray tracing algorithm as two points on a continuum that varies the number of rays that can be active in the system at once. In this view, our new algorithm generalizes both algorithms, selecting the appropriate point along the continuum based on the available resources of the host architecture. To demonstrate its feasibility, we sketch an implementation of our algorithm. Using detailed simulation of three scenes, we show that our algorithm obtains up to 32× reduction in the amount of geometry loaded when traversing primary rays, and up to 60× reduction when traversing shadow rays, with relatively little added overhead to handle ray state. We conclude that our notion of *dynamically scheduled rays* provides data access patterns that are spatially coherent both in terms of scene space and in terms of localized data access. Our algorithm can be easily combined with current ray tracing optimizations for coherent ray data, and, unlike those techniques, it promises to scale for use with complex lighting models.

The remainder of the paper is organized as follows: Section 2 describes our algorithm in detail and presents our implementation sketch; we describe our simulation and testing framework in Section 3; we discuss our results in Section 4; in Section 5 we present related work, and we sketch future directions and draw conclusions in Section 6.

*[pnav | fussell | lin]@cs.utexas.edu

2 Dynamic Ray Scheduling

The goal of our new ray tracing algorithm is to actively manage ray and geometry state to provide better cache utilization and lower

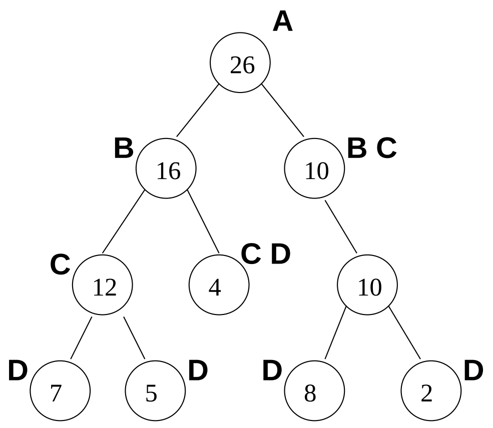

Figure 1: Queue Point Selection — Here, we demonstrate how our queue point selection algorithm works on a toy k-d tree. We measure the amount of cache available to hold geometry and determine what is the maximum amount of geometry (g_{max}) that can be loaded without exceeding available cache capacity. We select the first node on each branch of the tree that contains geometry: $g \leq g_{max}$. In this figure, if $g_{max} \geq 26$, the root (**A**) is the only queue point, and our algorithm degenerates to Whitted-style ray tracing, because all geometry fits in cache. If $26 > g_{max} \geq 16$, the internal nodes (**B**) are queue points. If $16 > g_{max} \geq 10$, the nodes (**C**) are queue points. If $g_{max} \leq 10$, the leaves (**D**) are queue points. Note that even if g_{max} is smaller than the amount of geometry at a leaf, that leaf is made a queue point because there is no remaining acceleration structure beneath it (see Section 2.2).

bandwidth requirements, which will in turn lead to faster execution time.

Our algorithm is rooted in two concepts: rays can be traced independently (non-recursively), and rays can be queued at regions in scene space where the geometry in that region fits completely in available memory. Taken together, these concepts permit tight control on the use of memory resources because, for any particular queue point, there is a known, tight upper bound on the amount of data that must be touched to process all the rays in that queue.

Our algorithm seeks to optimize both (1) bandwidth utilization between main memory and the lowest level of processor cache and (2) utilization of the lowest level of processor cache itself. Without loss of generality, we will refer to DRAM-to-L2 bandwidth and L2 utilization in our discussion, since these are common components of the multi-core hardware we target (see Figure 2).

The algorithm described here uses a k-d tree as the acceleration structure, but it could be adapted to other acceleration structures, including regular grids, hierarchical grids, and bounding volume hierarchies. The ability of the chosen acceleration structure to adapt to varying densities of scene geometry directly affects the quality of scheduling possible by determining how much flexibility we have in choosing queue points for rays. Our discussion will provide insight as to how the acceleration structure interacts with other parts of the algorithm, but a thorough analysis of the impact of acceleration structure choice is beyond the scope of this paper.

2.1 Traversal Algorithm

Our traversal algorithm traces rays from the root of the acceleration structure down to queue points, where further ray processing is deferred. It later iterates over these queue points to complete all ray traversals. To simplify our discussion, we first describe the traversal of primary rays only. Our technique, however, is applicable to all ray types, so we then generalize it to deal with secondary rays.

2.1.1 Traversing Primary Rays

We select queue points in the acceleration structure based on the amount of geometry that will fit in available cache. Each queue point is the root of a subtree of the acceleration structure, a subtree that contains no more geometry than will fit into L2 cache. See Figure 1 for an example. If the entire scene can fit into cache, then the root of the acceleration structure becomes the only queue point, and the traversal degenerates to Whitted's recursive algorithm.

Our algorithm can also efficiently schedule worst-case conditions in an acceleration structure. Sometimes the construction algorithm for the acceleration structure cannot adapt to dense local geometry. At such points in a k-d tree, a leaf with an unusually large amount of geometry is placed in the acceleration structure. If the geometry at that leaf exceeds cache capacity, a recursive ray traversal will *always* thrash the cache *each time* such a leaf is pierced by a ray. Our algorithm treats such leaves as a separate scheduling problem, and loads blocks of both rays and geometry to process the queue efficiently. See Section 2.2 for implementation details.

Our algorithm queues all primary rays, then iterates over the queues until all rays have terminated or have left the bounds of the scene. When a queue is selected for processing, each ray traverses any of the remaining subtree and is tested for intersection against the geometry at each leaf of the subtree that the ray may reach. Once a ray is selected at this stage, it is processed until either a successful intersection is found or until the ray leaves the bounds of the subtree. If the ray leaves the bound of the subtree, it continues its traversal through the full acceleration structure, either to the next queue point or until it leaves the bounds of the scene.

When a ray intersects a surface, we can either shade the intersection point immediately (as in ray casting) or save it for deferred casting of secondary rays. A pixel id is maintained with the ray so that the proper pixel can be shaded. When supersampling, samples can be blended in the framebuffer as they arrive. If secondary rays are cast (described in Section 2.1.2), then the point is shaded iteratively as each secondary ray is processed.

2.1.2 Traversing Secondary Rays

Our algorithm can be easily generalized to handle secondary rays. These rays are processed in generations: shadow rays from the current generation are processed, then any newly spawned non-shadow rays are processed. By processing rays in generations, we limit the amount of active ray state in the system while still providing coherent access to scene geometry.

To generate shadow rays and other secondary rays, we maintain the intersection points for the current generation of rays. For each point light, we trace shadow rays from the light toward the intersection points, which makes the traversal identical to the primary ray traversal method described in Section 2.1.1. Shadow rays inherit both the pixel id and the shading information from their spawning ray. Thus, when light visibility has been determined, the shading contribution, if any, can be added to the appropriate pixel.

Once all shadow rays for the current generation have terminated, we traverse newly spawned non-shadow rays. Each new ray starts queued at whichever queue point contains its origin. Our algorithm then iterates over queue points to traverse rays, as before. Note that while we may achieve less coherence here than for primary and shadow rays, we can achieve significantly better coherence than a recursive ray tracer by allowing many secondary rays to be active at once. Once all rays of this new generation have been processed, any resultant intersection points are used to generate the next generation of shadow and secondary rays. This process continues until no new secondary rays are generated.

Note that our technique can employ adaptive sampling techniques by maintaining ray information across generations. We do this by adding a field in the ray structure for a pointer to the information to be maintained (see Figure 2. This solution is similar to

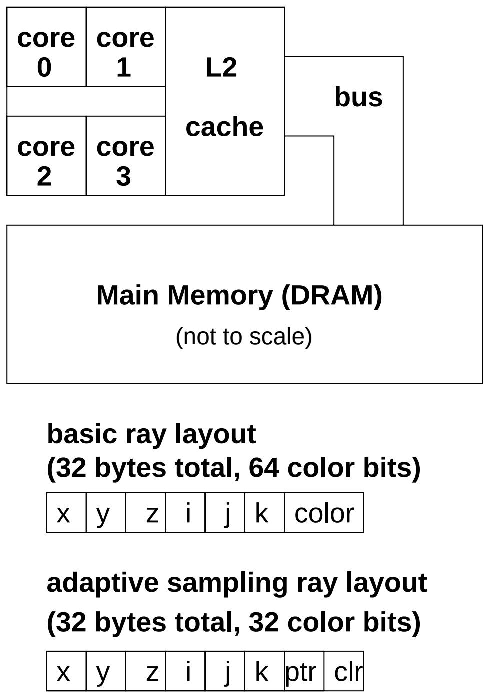

Figure 2: System Block Diagram and Ray Layout — we target a multi-core architecture, as represented in this block diagram. We use a tight ray representation, whether or not adaptive sampling is required. Our basic ray layout supports 48-bit color plus a 16-bit alpha channel. Our adaptive sampling ray layout, which contains a pointer to the information that must be maintained for the adaptive sampler, supports 24-bit color plus an 8-bit alpha channel.

splitting shading information for geometry into a separate structure, which is loaded only when needed.

2.2 Implementation Sketch

We now describe how our algorithm can be implemented on a modern multi-core processor. We maintain geometry and acceleration structure data in cache while buffering rays to ensure threads are maximally occupied. For the discussion below, we assume a 4MB L2 cache.

We want the acceleration structure to remain resident in cache. We represent k-d tree nodes using eight bytes, similar to the k-d tree used in PBRT[PH04]. We use an additional bit from the least-significant end of the mantissa of the split location in order to indicate whether a node is a queue point (leaving 20 bits for the single-precision mantissa representation). We expect this quantization not to significantly affect the quality of the k-d tree. Using this representation, 128K nodes can remain resident if we reserve 1MB of the L2 for nodes. If the k-d tree is larger, we have the option of reserving more space or maintaining only the top of the tree, from the root down to the queue points. If we maintain only the top of the tree, we must load each subtree before processing its associated queue. This situation will only occur for extremely large scenes, where the added cost for loading the subtree will be insignificant compared to the cost of loading the associated geometry.

We also maintain a table that associates each queue point with a buffer in main memory that contains the actual ray queue. We keep this table and its associated buffers in memory so that rays can be enqueued quickly and without having to load data to compute the address to which the ray should be sent. This table costs 8 bytes per entry, and we expect 32K queue points to be sufficient for most trees, which makes the table cost 256KB.

We must have rays cached to perform traversals and intersections, yet we expect to have hundreds to thousands of rays queued at each point. Bringing all rays in at once would evict other needed data from cache. Further, we do not need all rays loaded, since we can only process as many rays as there are threads available. Yet we require more than a single ray per thread so that the thread can swap if a ray reaches a leaf with yet-uncached geometry. We want to buffer enough rays to mask the latency of the initial cache miss on a leaf's geometry. We know that all queued rays must be traversed through the active subtree, so this work will be available so long as there are queued rays. A single ray traversal step in a k-d tree is a ray-plane intersection test, made simpler because the plane is guaranteed to be axis-aligned. The ray-plane intersection test can be computed with a multiply, an add and a comparison. With instruction latency, the test takes about seven cycles to complete[SSM*05]. We expect about ten traversal steps will be necessary to take a ray from the queue point to a leaf, and modern DRAMs can return a random data request in about 80 cycles. Therefore, having two rays per thread should be sufficient. We represent our rays in 32 bytes (see Figure 2), and if we have 4 threads, the ray buffer requires 256 bytes.

Finally, we need to cache geometry. We select queue points in the acceleration structure so that the geometry in the subtree will fit in available cache (taking into account the acceleration structure, ray buffer, etc., described above). We could load all geometry in the subtree immediately, but we do not yet know which geometry, if any, will actually be required. To avoid spurious geometry loads, we wait to load geometry until a ray has definitely reached it (i.e., reached the leaf that contains the geometry). Note that if a queue point is at a leaf, then the geometry may be loaded immediately because the geometry will definitely be tested for intersection.

For systems where cache resources are very limited, it is possible to have a queue point, which must be at a leaf of the acceleration structure, that contains more geometry than will fit in available cache. A Whitted-style ray tracer will thrash the cache *each time* a ray pierces such a leaf. Our algorithm permits more flexible approaches by treating this condition as a separate scheduling problem. We know the amount of geometry at each leaf from building the acceleration structure, so we can detect a thrashing condition before loading any geometry. We create cache-sized blocks of geometry and iterate over them as we test for intersection against the rays in the ray buffer. This iterated processing technique results in fewer overall cache loads than allowing uncontrolled cache thrashing.

3 Experimental Methodology

We have performed a feasibility study of our algorithm using a research (non-optimized) ray tracer with a simulated L2 cache. With this arrangement, we test the performance of our algorithm in terms of cache utilization and bandwidth consumption, over a range of cache sizes to determine its effectiveness. Our simulation results, presented here, are promising enough that we are currently creating an optimized implementation targeted for specific hardware. We discuss this implementation further in Section 6.

To obtain the cache measurements in our simulation, we create a memory trace using explicit reads and writes in our code and run that trace through various cache configurations using Dinero IV[EH98], a light-weight trace-driven simulator. We simulate cache sizes in power-of-two increments from 1KB to 4MB, with 64B cache lines. This wide range of sizes allows us to understand the performance of our algorithm both when cache resources are scarce and when they are plentiful. We make each cache fully associative to eliminate conflict misses. Thus, after the cache is warm, all misses are capacity misses.

We test our algorithm on three scenes, each rendered at 1024×1024 resolution (see Figure 3). These models provide a variety

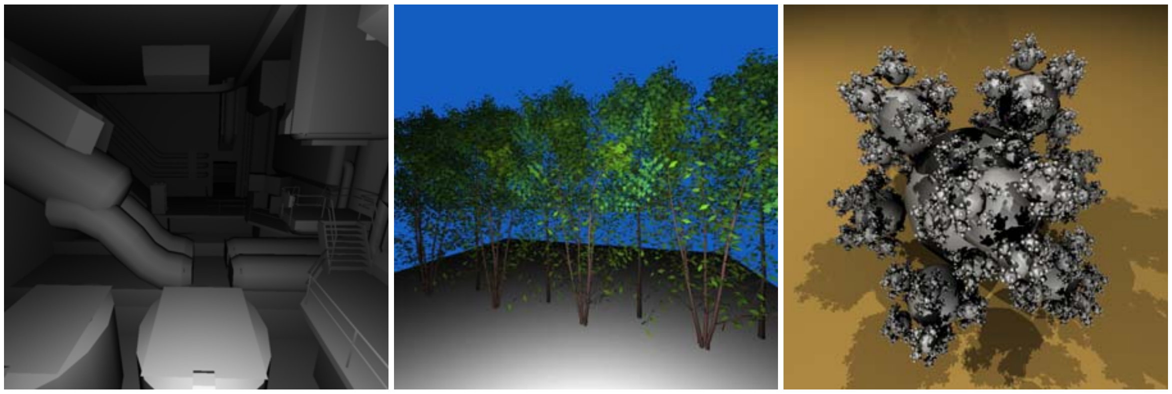

Figure 3: Test scene images — We test our algorithm on three scenes: `room`, `grove`, and `sphereflake`. For each we measure the total geometry in the scene and the number of triangles *potentially visible* (p-v), which must be tested for intersection when tracing primary rays only and primary + secondary rays. [`room`: 47K triangles, 6.6K p-v primary, 7.7K p-v secondary]; [`grove`: 164K triangles, 127K p-v primary, 142K p-v secondary]; [`sphereflake`: 797K triangles, 258K p-v primary, 535K p-v secondary]. Note that the geometry artifacts in `room` are contained in the scene specification and are not due to our ray tracer.

of total geometry, potentially-visible (p-v) geometry, and geometric topology. We use a small architectural scene (`room`: 47K triangles, 6.6K p-v primary, 7.7K p-v secondary), a grove of tree models (`grove`: 164K triangles, 127K p-v primary, 142K p-v secondary), and a five-level sphereflake model from the Standard Procedural Database scenes[Hai87] (`sphereflake`: 797K triangles, 258K p-v primary, 535K p-v secondary). We specifically mention potentially-visible geometry when tracing primary and secondary rays because these figures are a more accurate measure of the geometry load when rendering. Total geometry affects the size and quality of the acceleration structure and whether the scene can fit in main memory, but it does not impact the geometry traffic between main memory and processor cache unless all geometry must be tested for intersection.

We compare our algorithm against two recursive ray tracers: a single-ray tracer using rays ordered along a Hilbert curve, and a ray packet tracer using 8×8 packets tiled over the image plane. We use a Hilbert curve ordering for single rays, since this ordering is known to produce better utilization of the memory system[Voo91]. We use an 8×8 packet size since it is the midpoint of currently popular packet sizes (4×4[RSH05, WIK*06] — 16×16[WBS07]) and it was recently found to provide the most system speed-up in this range[BEL*07].

4 Results and Discussion

We present the results of our algorithm feasibility study here. We show that by using dynamic ray scheduling to actively manage both ray and geometry data, we can improve the cache utilization for geometry data by as much as 32× over recursive ray tracing when tracing primary rays, and by as much as 60× when tracing both primary and point-light shadow rays. These savings will become increasingly important as additional lighting and shading data is stored in scene space, both increasing the amount of data that must be loaded and decreasing the available cache space for geometry.

4.1 Tracing Primary Rays

In Figures 4–6, we present our measurements for tracing primary rays only. These measurements show that our algorithm reduces geometry traffic between DRAM and L2 for all cache sizes at the cost of increased ray traffic. Ray traffic is more desirable than geometry traffic, since a thread must block for a geometry load but can switch to another ray if one is available. Said another way, we want to keep geometry in cache and stream rays, so long as there are enough rays in cache to keep all threads busy.

Further, our algorithm significantly reduces geometry traffic *when system resources are scarce*. When the data load on the system is greatest, our algorithm adapts to make efficient use of available resources. Recursive ray tracing cannot adapt in this way, and ends up thrashing the cache with geometry data. Note that recursive ray tracing maximally constrains the amount of *ray traffic* at the potential cost of increased geometry traffic. Our algorithm relaxes this constraint, allowing ray traffic to grow while significantly reducing geometry traffic. Thus our algorithm can make efficient use of system resources and adapt to various system loads.

For example, the Intel 5000X chipset[Int07] provides 21GB/s peak bandwidth between DRAM and L2. At 20 frames per second (fps), a ray tracer has only enough time to use 1GB of bandwidth for any one frame (we are ignoring computation time here which would further reduce the time available to load data). When cache resources are scarce, our algorithm reduces geometry traffic to fit within this bandwidth constraint. When cache resources are plentiful, the ray traffic overhead generated by our algorithm does not exceed available bandwidth.

4.2 Tracing Secondary Rays

Our algorithm performs well when tracing both primary rays and secondary rays. In this section, we present measurements for tracing primary rays and shadow rays from three point-lights in each scene. We present our measurements for each light individually, to observe the effects of each light position, and for all three lights together, to observe their interaction. Because our algorithm does not always reduce geometry traffic in our secondary ray tests, and as we have said, we always have ray overhead, we focus our discussion on geometry traffic and what our measurements imply for more complex lighting models.

In Figures 7–9, we show how much our algorithm reduces geometry traffic when tracing both primary and point-light shadow rays. The relative performance of our algorithm depends on the number and location of point lights in the scene. In Figures 7 and 8, light 0 is located at the camera point, which is a best-case for recursive algorithms because no new geometry is accessed between the hit point and the light. Thus, if cache is sufficiently large enough to hold the geometry tested for intersection against the primary ray, then no new geometry will be loaded. Our algorithm does comparatively worse because we trace all primary rays, then trace the shadow rays. Thus we miss this locality. However, this relative measure does not translate into negative system performance because it only occurs when resources are plentiful. As we discuss in `room`

cache	geometry traffic			ray traffic			geometry traffic reduction		total traffic reduction	
size	recursive	packet	dynamic	recursive	packet	dynamic	recursive	packet	recursive	packet
1 K	6056.65	—	4531.30	32.00	—	245.32	33.7%	—	27.5%	—
2 K	5863.94	—	943.52	32.00	—	323.14	521.5%	—	365.5%	—
4 K	4939.97	403.16	231.89	32.00	32.00	313.38	2030.3%	73.9%	811.8%	-25.3%
8 K	1269.55	136.71	90.05	32.00	32.00	314.44	1309.8%	51.8%	221.8%	-139.8%
16 K	133.23	18.58	3.95	32.00	32.00	285.63	3274.7%	370.7%	-75.3%	-472.5%
32 K	4.86	11.10	3.31	32.00	32.00	263.97	46.9%	235.5%	-625.1%	-520.1%
64 K	3.83	8.75	2.90	32.00	32.00	231.63	32.3%	202.1%	-554.5%	-475.5%
128 K	3.45	6.37	2.74	32.00	32.00	155.70	26.1%	132.6%	-346.9%	-312.9%
256 K	3.24	3.14	2.57	32.00	32.00	108.39	26.4%	22.4%	-214.8%	-215.8%
512 K	3.05	2.90	2.53	32.00	32.00	72.49	20.6%	14.5%	-114.0%	-115.0%
1024 K	2.89	2.77	2.42	32.00	32.00	52.53	19.2%	14.6%	-57.5%	-58.0%
2048 K	2.59	2.55	2.42	32.00	32.00	32.35	7.3%	5.4%	-0.5%	-0.6%
4096 K	2.42	2.42	2.42	32.00	32.00	32.00	0.0%	0.0%	0.0%	0.0%

Figure 4: Data traffic (MB) for `room` — Even on our smallest test scene, both in total geometry and in visible geometry, our algorithm reduces geometry traffic. The most dramatic traffic reduction comes at the smallest tested cache sizes. While overall traffic increases significantly for the middle range of caches (32K - 512K), we expect this not to impact performance because our algorithm prevents cache pollution from ray data and the traffic does not exceed peak bandwidth on current architectures. Note that ray traffic does not decrease monotonically due to the selection of different queue points in the acceleration structure. We do not report results for packets for 1K and 2K caches because they are too small to contain the packet and any geometry.

`grove`

cache	geometry traffic			ray traffic			geometry traffic reduction		total traffic reduction	
size	recursive	packet	dynamic	recursive	packet	dynamic	recursive	packet	recursive	packet
1 K	11084.02	—	9641.98	32.00	—	390.63	15.0%	—	10.8%	—
2 K	10230.37	—	5992.17	32.00	—	382.68	70.7%	—	61.0%	—
4 K	9132.21	4120.02	2564.29	32.00	32.00	340.52	256.1%	60.7%	215.5%	42.9%
8 K	7861.03	1925.96	943.50	32.00	32.00	294.80	733.2%	104.1%	537.4%	58.1%
16 K	5361.92	781.82	227.31	32.00	32.00	251.28	2258.9%	243.9%	1027.0%	70.0%
32 K	1313.46	365.39	139.96	32.00	32.00	211.86	838.5%	161.1%	282.4%	13.0%
64 K	182.14	216.75	118.75	32.00	32.00	177.97	53.4%	82.5%	-38.6%	-19.3%
128 K	138.09	169.63	105.55	32.00	32.00	153.36	30.8%	60.7%	-52.2%	-28.4%
256 K	121.41	168.35	97.31	32.00	32.00	127.47	24.8%	73.0%	-46.5%	-12.2%
512 K	112.68	166.10	91.95	32.00	32.00	101.35	22.5%	80.6%	-33.6%	2.5%
1024 K	107.00	160.23	87.37	32.00	32.00	86.23	22.5%	83.4%	-24.9%	10.7%
2048 K	103.40	141.21	84.92	32.00	32.00	70.06	21.8%	66.3%	-14.5%	11.8%
4096 K	100.44	95.67	83.87	32.00	32.00	57.78	19.8%	14.1%	-7.0%	-10.9%

Figure 5: Data traffic (MB) for `grove` — On this larger test scene, the benefit of our algorithm becomes clear. When cache resources are scarce (here, ¡ 64K), our algorithm significantly reduces data traffic. When cache resources are plentiful (here, $\geq$ 64K) we still obtain better cache utilization with respect to geometry, at the cost of more ray traffic. Again, we expect this increased ray traffic to not affect system performance because peak bandwidth between DRAM and cache is not exceeded. We do not report results for packets for 1K and 2K caches because they are too small to contain the packet and any geometry.

`sphereflake`

cache	geometry traffic			ray traffic			geometry traffic reduction		total traffic reduction	
size	recursive	packet	dynamic	recursive	packet	dynamic	recursive	packet	recursive	packet
1 K	4185.85	—	3451.22	32.00	—	142.80	21.3%	—	17.4%	—
2 K	3600.98	—	2263.18	32.00	—	133.51	59.1%	—	51.6%	—
4 K	2773.87	922.74	629.53	32.00	32.00	130.79	340.6%	46.6%	269.0%	25.6%
8 K	1289.94	378.95	208.70	32.00	32.00	123.86	518.1%	81.6%	297.5%	23.6%
16 K	339.72	205.01	110.64	32.00	32.00	132.35	207.0%	85.3%	53.0%	-2.5%
32 K	122.87	134.69	78.10	32.00	32.00	128.85	57.3%	72.5%	-33.6%	-24.2%
64 K	97.58	108.71	73.60	32.00	32.00	134.33	32.6%	47.7%	-60.5%	-47.8%
128 K	88.69	105.91	70.52	32.00	32.00	137.08	25.8%	50.2%	-72.0%	-50.5%
256 K	83.53	105.23	68.53	32.00	32.00	141.08	21.9%	53.5%	-81.4%	-52.8%
512 K	80.42	103.85	67.63	32.00	32.00	149.31	18.9%	53.6%	-93.0%	-59.7%
1024 K	78.44	91.00	66.80	32.00	32.00	142.11	17.4%	36.2%	-89.2%	-69.8%
2048 K	77.09	74.84	66.04	32.00	32.00	146.63	16.7%	13.3%	-94.9%	-99.0%
4096 K	76.37	74.82	65.70	32.00	32.00	150.94	16.2%	13.9%	-99.9%	-102.8%

Figure 6: Data traffic (MB) for `sphereflake` — This scene has more total geometry, but less potentially visible geometry than `grove`. The acceleration structure created for this scene challenges our algorithm. The geometry is finely tessellated yet structured, which results in a deep, poor k-d tree. Our algorithm still reduces geometry traffic for all caches. Note that the effect of the acceleration structure can be seen in the parabolic trend of the ray traffic measurements. We do not report results for packets for 1K and 2K caches because they are too small to contain the packet and any geometry.

`room` **with shadows**

cache	light 0 traffic reduction		light 1 traffic reduction		light 2 traffic reduction		all 3 lights traffic reduction	
size	recursive	packet	recursive	packet	recursive	packet	recursive	packet
1 K	26.6%	—	-2.6%	—	-9.1%	—	-7.9%	—
2 K	438.4%	—	300.6%	—	231.5%	—	240.0%	—
4 K	1404.8%	930.9%	1450.4%	360.4%	1263.5%	304.6%	1060.9%	739.8%
8 K	756.4%	571.3%	1754.8%	211.2%	1704.5%	203.8%	1148.7%	456.7%
16 K	1588.2%	1738.4%	5927.2%	607.7%	5966.1%	618.3%	3368.8%	1380.2%
32 K	-33.1%	76.6%	73.8%	221.0%	69.2%	212.2%	-11.7%	65.1%
64 K	-48.8%	57.0%	24.2%	219.6%	23.9%	218.5%	-57.8%	64.8%
128 K	-56.8%	19.5%	6.4%	166.8%	10.8%	177.7%	-81.1%	38.8%
256 K	-56.5%	-60.5%	-0.8%	63.3%	2.9%	68.6%	-95.0%	-17.3%
512 K	-64.2%	-72.9%	-12.6%	-9.3%	-7.6%	-4.2%	-116.4%	-110.7%
1024 K	-65.9%	-72.8%	-17.3%	-22.2%	-10.9%	-15.5%	-125.2%	-134.0%
2048 K	-84.4%	-87.4%	-39.1%	-42.3%	-31.7%	-34.8%	-166.1%	-172.8%
4096 K	-97.6%	-97.6%	-55.1%	-55.1%	-46.9%	-46.9%	-197.9%	-197.9%

Figure 7: Data traffic (MB) for `room` with shadows — On this smallest scene, the worst for our algorithm because total geometry can nearly fit in cache, our algorithm only reduces geometry traffic when cache resources are scarce. Light 0 is located at the camera point, which is a best-case for recursive algorithms since no new geometry is accessed by the shadow ray. We do not report results for packets for 1K and 2K caches because they are too small to contain the packet and any geometry.

`grove` **with shadows**

cache	light 0 traffic reduction		light 1 traffic reduction		light 2 traffic reduction		all 3 lights traffic reduction	
size	recursive	packet	recursive	packet	recursive	packet	recursive	packet
1 K	11.0%	—	10.3%	—	-21.7%	—	-12.4%	—
2 K	61.8%	—	73.3%	—	23.9%	—	37.5%	—
4 K	209.3%	114.7%	273.8%	109.9%	155.8%	26.3%	178.1%	101.2%
8 K	513.7%	277.2%	755.6%	230.6%	477.7%	75.2%	475.7%	263.3%
16 K	1182.0%	729.1%	1934.7%	516.9%	1407.4%	238.0%	1095.7%	640.3%
32 K	346.1%	315.7%	955.0%	311.4%	604.3%	164.0%	510.1%	410.1%
64 K	-21.9%	0.1%	155.0%	153.4%	71.5%	76.2%	90.9%	127.2%
128 K	-42.8%	-16.2%	65.7%	97.7%	22.5%	44.7%	20.8%	35.8%
256 K	-50.2%	-8.3%	34.9%	88.6%	8.1%	46.6%	-5.3%	22.7%
512 K	-53.3%	-4.0%	23.2%	108.6%	1.5%	58.8%	-21.3%	30.7%
1024 K	-53.7%	-2.7%	10.9%	115.6%	-5.4%	63.6%	-39.8%	34.1%
2048 K	-54.9%	-13.4%	1.6%	113.6%	-9.7%	64.8%	-56.6%	36.0%
4096 K	-57.6%	-65.4%	-3.5%	96.7%	-16.2%	38.3%	-74.0%	29.4%

Figure 8: Data traffic (MB) for `grove` with shadows — On this larger scene, our algorithm reduces geometry traffic both when cache is scarce and when cache is plentiful. Light 0 is located at the camera point, which is a best-case for recursive algorithms since no new geometry is accessed by the shadow ray. We do not report results for packets for 1K and 2K caches because they are too small to contain the packet and any geometry.

`sphereflake` **with shadows**

cache	light 0 traffic reduction		light 1 traffic reduction		light 2 traffic reduction		all 3 lights traffic reduction	
size	recursive	packet	recursive	packet	recursive	packet	recursive	packet
1 K	-1.6%	—	-2.4%	—	5.1%	—	-4.2%	—
2 K	21.6%	—	27.0%	—	41.6%	—	27.3%	—
4 K	228.4%	73.1%	304.0%	109.7%	332.7%	130.2%	306.6%	160.7%
8 K	421.9%	128.7%	553.8%	179.7%	580.1%	207.3%	631.7%	243.8%
16 K	275.5%	152.8%	298.5%	160.1%	340.8%	185.8%	577.2%	271.5%
32 K	61.7%	82.8%	84.1%	102.5%	83.0%	103.0%	148.9%	184.7%
64 K	20.0%	33.7%	41.2%	52.2%	50.6%	61.1%	72.0%	96.6%
128 K	7.6%	25.8%	28.4%	38.9%	37.3%	51.1%	47.9%	65.5%
256 K	-0.8%	26.3%	22.3%	42.3%	29.9%	55.1%	34.6%	65.6%
512 K	-6.4%	27.2%	18.1%	44.8%	25.2%	58.7%	26.2%	69.9%
1024 K	-11.3%	24.0%	14.5%	43.1%	21.1%	58.0%	19.9%	68.1%
2048 K	-14.3%	-7.6%	12.6%	21.0%	17.7%	32.7%	14.5%	64.1%
4096 K	-17.4%	-18.7%	8.6%	4.2%	16.7%	12.4%	9.6%	16.5%

Figure 9: Data traffic (MB) for `sphereflake` with shadows — On this large scene, as on `grove`, our algorithm reduces geometry traffic both when cache is scarce and when cache is plentiful. Light 0 is located near the camera point, which is a good case for recursive algorithms since little new geometry is accessed by the shadow ray. We do not report results for packets for 1K and 2K caches because they are too small to contain the packet and any geometry.

Section 4.1, so long as our algorithm does not exceed peak bandwidth, overall rendering time will not be affected. Further, recursive algorithms cannot maintain this locality under a more complex lighting model. Global illumination approximations generate tens to hundreds of secondary rays per primary ray, each of which may access new, uncached geometry. A recursive ray tracer will eventually thrash the cache with geometry data, whereas our algorithm will continue to process these rays coherently.

Our algorithm performs slightly worse for the 1K cache, the smallest we use. We attribute this to the difference in shadow ray direction between our algorithm and the recursive algorithms. Our algorithm traces shadow rays from point light to hit points, whereas the recursive algorithms trace shadow rays from hit points to point light. In so doing, the recursive algorithms gain a bit of temporal locality by immediately testing the shadow ray for intersection with the geometry around the primary ray hit point. For larger cache sizes, the ray coherence benefits of tracing shadow rays from light to hit point overwhelm this smaller effect.

5 Related Work

As mentioned earlier, our work builds on several previous ray scheduling schemes to create a ray tracing algorithm in which both geometry and rays are actively managed components. Below, we discuss this and other related work to better distinguish the contributions of our algorithm.

All recursive ray tracers implement some form of Whitted's original recursive ray tracing algorithm[Whi80]. Recent innovations to the basic algorithm include the use of SIMD instructions to trace multiple rays in parallel[WSBW01] and tracing a frustum of rays through the acceleration structure to eliminate upper level traversal steps[RSH05]. These techniques can increase the number of rays active in the system at once, but none of them allow rays to be dynamically scheduled: once a ray (or packet of rays) is selected for traversal, the selected ray(s) *and all child rays* must be traced to completion before another selection choice is made. The fixed active ray state in these algorithms hampers their ability to handle system-related issues such as thrashing and effectively hiding latency. Recently, Boulos, et al.[BEL*07], attempted to achieve real-time frame rates on a current workstation-class system for both Whitted-style ray tracing with full specular effects and distribution ray tracing[CPC84]. They claim interactive rates for their Whitted-style tracer for both primary and secondary rays by employing clever methods of organizing secondary rays. Further, Boulos, et al., observes that shading computation may soon overtake visibility computation as the primary system cost. Note that our algorithm could accept a modified form of their secondary ray organization technique. Also, because our algorithm operates in local regions of scene space, it facilitates grouping similarly shaded points, making efficient use of shading computation.

Pharr, et al.[PKGH97], take an approach quite different from the Whitted tracing model. They decompose the rendering equation[Kaj86] so that it can be calculated forward as rays are traced, rather than using the recursion stack to accumulate shading calculations. Pharr's algorithm targets efficient use of main memory and traffic between disk and RAM to render models that cannot fit in main memory. Pharr uses Monte-Carlo-based global illumination to model indirect diffuse lighting, a technique that uses many sample rays per primary ray. Since this algorithm targeted a higher level of the memory hierarchy, there are aspects of it that are poorly suited for RAM to cache traffic. The acceleration structure is a uniform grid, the non-adaptivity of which makes it more difficult to guarantee the amount of geometry found at any particular cell. This variance may be masked at the RAM level, but at the cache level it complicates effective scheduling. Also, while rays can be reordered under Pharr's algorithm, ray state is not actively managed. Pharr actively seeks to get deep into the ray tree quickly, the result of which is that many secondary rays of many ray generations are active at once in the system. Again, while this technique may be effective when considering disk to RAM traffic, the ray state explosion that results can cause uncontrollable thrashing in cache-sized memories. Our technique controls both geometry state and ray state to ensure efficient operation within a given system.

We are aware of two implementations of algorithms similar to that of Pharr et al. Dachille and Kaufman[DK00] used Pharr's ray deferring algorithm in a hardware-based volume rendering system. They use specialized hardware to perform standard volume rendering and volume rendering with a simplified global illumination model. In this system, rays are collected at each cell of the volume, much like how rays collect in Pharr's uniform grid. Cells are then scheduled for processing like Pharr's algorithm. They were able to achieve interactive frame rates on a simulation of their hardware. Because they use direct volume rendering, there is no significant geometry traffic per cell: the system loads the eight vertices of the cell and trilinearly interpolates each sample along each ray. Thus, their system does not address managing geometry traffic at all. Steinhurst, et al.[SCL05], use Pharr-like reordering to obtain better cache performance for photon mapping. Like Pharr, their system experiences ray state explosion. However, its effects cannot be masked at the cache level, and the performance of their system suffers. We expect that our algorithm would perform better on this task because it actively manages ray state.

6 Conclusions and Future Work

We have presented a ray tracing algorithm that dynamically schedules rays in order to actively manage both ray and geometry data. In so doing, our algorithm significantly reduces geometry traffic between DRAM and L2 cache with a moderate increase in ray traffic. Further, our algorithm dramatically reduces overall data traffic *when memory resources are scarce*, thus permitting the efficient ray tracing of large, complex scenes.

We have three areas for future work. First, we will implement a cycle-accurate cache model in our simulator to obtain measurements for the utilization analysis we give in Section 4.1. Second, we will expand our research ray tracer to include secondary rays, global illumination and distribution ray tracing effects[CPC84]. Third, we will complete the implementation of our system as described in Section 2.2 targeted at the latest multi-core multi-threaded commodity architecture. With this implementation, we hope to either achieve real-time ray tracing performance directly or to determine what additional system resources are required to achieve it.

Acknowledgments

Thanks to the Real-Time Graphics and Systems group at UT for their helpful comments. This research was supported in part by the National Science Foundation under CISE Research Infrastructure Grant EIA-0303609.

References

[BEL*07] Boulos S., Edwards D., Lacewell J. D., Kniss J., Kautz J., Wald I., Shirley P.: Packet-based Whitted and Distribution Ray Tracing. In *Proceedings of Graphics Interface 2007* (2007).

[CPC84] Cook R. L., Porter T., Carpenter L.: Distributed ray tracing. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1984), ACM Press, pp. 137–145.

[DK00] Dachille IX F., Kaufman A.: Gi-cube: an architecture for volumetric global illumination and rendering. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 2000), ACM Press, pp. 119–128.

[EH98] Edler J., Hill M. D.: Dinero IV cache simulator (http://www.cs.wisc.edu/m̃arkhill/DineroIV/), 1998.

[Hai87] HAINES E.: A proposal for standard graphics environments. *IEEE computer graphics and applications* (November 1987), 3–5. http://www.acm.org/tog/resources/SPD/.

[Int07] INTEL CORPORATION: Intel 5000X Chipset Overview (http://www.intel.com/products/chipsets/5000x), 2007.

[Jen96] JENSEN H. W.: Global Illumination using Photon Maps. in X. Pueyo and P. Schröder, editors, Rendering Techniques '96, pages 21–30. Springer-Verlag, 1996.

[Kaj86] KAJIYA J. T.: The rendering equation. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1986), ACM Press, pp. 143–150.

[PH04] PHARR M., HUMPREYS G.: *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 2004.

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering complex scenes with memory-coherent ray tracing. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), ACM Press/Addison-Wesley Publishing Co., pp. 101–108.

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. In *SIGGRAPH '05: Proceedings of the 32nd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2005), ACM Press.

[SCL05] STEINHURST J., COOMBE G., LASTRA A.: Reordering for cache conscious photon mapping. In *GI '05: Proceedings of the 2005 conference on Graphics interface* (School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2005), Canadian Human-Computer Communications Society, pp. 97–104.

[SSM*05] SLUSALLEK P., SHIRLEY P., MARK W., STOLL G., WALD I.: Parallel & distributed processing. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), ACM Press, p. 11.

[SWW*04] SCHMITTLER J., WOOP S., WAGNER D., PAUL W. J., SLUSALLEK P.: Realtime ray tracing of dynamic scenes on an FPGA chip. In *Graphics Hardware 2004* (2004).

[Voo91] VOORHIES D.: Space-filling curves and a measure of coherence. in J. Arvo, editor, Graphics Gems II, pages 26–30. Academic Press, 1991.

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (2007), 6.

[Whi80] WHITTED T.: An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (June 1980), 343–349.

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics* (2006), 485–493. (Proceedings of ACM SIGGRAPH 2006).

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive rendering with coherent ray tracing. In *Proc. of Eurographics 2001* (2001).

[WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: a programmable ray processing engine. In *SIGGRAPH '05: Proceedings of the 32nd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2005), ACM Press.