The Dissertation Committee for Paul Arthur Navrátil
certifies that this is the approved version of the following dissertation:

Memory-Efficient, Scalable Ray Tracing

Committee:

Donald S. Fussell, Supervisor

Calvin Lin, Supervisor

Keshav Pingali

Kelly Gaither

William R. Mark

Memory-Efficient, Scalable Ray Tracing

by

Paul Arthur Navrátil, B.A., B.S., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2010

Success in most things comes not from some gigantic stroke of fate, but from simple, incremental progress.

— Andrew Wood

Golden turtles always finish.

Acknowledgments

Thanks to all those who influenced my work and my personal and professional development during my graduate studies. Many thanks to my advisors Don Fussell and Calvin Lin for their unflagging support of my work, and to Bill Mark for his continued advice and support both at UT and at Intel. Special thanks to Hank Childs at UC-Davis and Lawrence Berkeley National Laboratory for his continued support, his encyclopedic knowledge of VisIt and VTK, and his rapid coding to create a platform for my work in the VisIt source.

Thanks to all the staff at the Texas Advanced Computing Center for providing an excellent environment for work and research. Special thanks to Kelly Gaither and Jay Boisseau for their enthusiastic support of my research. Thanks to Karl Schulz, Tommy Minyard, and Bill Barth for helpful discussions about efficient large-scale computing. Thanks to the Visualization group at TACC, particularly Greg P. Johnson for leading the World Community Grid discussions.

Thanks to the Real-Time Graphics and Parallel Systems group at UTCS for the many discussions about high-performance ray tracing. Particular thanks to Greg S. Johnson, Peter Djeu, Warren Hunt, Jeff Diamond and Gilbert Bernstein for their insightful questions and comments.

Thanks to the Graphics group at Intel for their support of my work. Special thanks to Alex Reshetov for allowing us to use the MLRTA ray tracer in our band-

width study. Thanks to Al McPherson, Pat McCormick, and the Visualization group at Los Alamos National Laboratory for their early and continued support. Thanks to Bruce Porter at UTCS for supporting my undergraduate work and for providing my initial opportunity in the graduate program. Thanks to Dan Miranker and Lance Obermeyer from Liaison Technology for their support and my initial opportunity in applied research. Thanks to Tim Purcell for supplying the tree models used in our **grove** scene.

Many thanks to the friends and family whose love and support made this journey possible. Thanks to Master John Blankenship and the wonderful family at Blankenship Martial Arts for teaching me dedication and perseverance. Thanks to Mike Matthews and the students at the Ashtanga Yoga Center of Austin for providing a space for centering and self-discovery. Thanks to all the Hill Country Trail Runners for many joyful romps through the woods. And thanks to Mom, Dad, Marcia, John, Eric, Laura, Joel, Coral, Diana, Tom, Sir Geoffie, Bob, Tony, Rosie, Gina, Nguyen, Edy, Mychal, Moody, Katie, Craig, Victor, Vliet, Michael, Jessica, Salem, Fu, and Bella. Love you all.

This research was funded in part by National Science Foundation grants ACI-9984660, EIA-0303609, ACI-0313263, CCF-0546236, OCI-0622780, OCI-0726063, and OCI-0906379; an Intel Research Council grant; and an IC^2 Institute fellowship.

— *PAN*

Memory-Efficient, Scalable Ray Tracing

Publication No. _____

Paul Arthur Navrátil, Ph.D.
The University of Texas at Austin, 2010

Supervisors: Donald S. Fussell
Calvin Lin

Ray tracing is an attractive rendering option because it can produce high quality images that faithfully represent physically-based phenomena. Its embarrassingly parallel nature makes it a natural choice for rendering large-scale scene data, especially on machines that lack specialized graphics hardware. Unfortunately, the traditional recursive ray tracing algorithm is exceptionally memory inefficient for large scenes, especially when using a shading model that generates incoherent secondary rays. Queueing ray tracers have been shown to control scene state under these conditions, but they allow ray state to grow unchecked. Instead, we propose a ray tracing framework that controls both ray and scene state by dynamically adjusting the rendering algorithm to meet memory requirements. Our *dynamic scheduling* framework generalizes recursive and queueing tracers into a spectrum of *ray schedules* that can vary the active amount of ray and scene data in order to match the characteristics of the hardware's memory system.

This dissertation describes our dynamic ray scheduling approach that operates on memory-bound work units, which consist of both rays and scene data.

It builds these work units by tracing rays iteratively and queueing them in spatial regions along with nearby data. By *dynamically scheduling* these work units, our approach can reduce data loads and improve total runtime by $2\times$ to $30\times$. In addition, we show that our algorithm scales across more than 1000 distributed processors, which is an order of magnitude larger than previously published results. Our approach enables the use of complex lighting models on large data, particularly scientific data, which improves image quality and thereby improves the scientific insights possible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Humans are visual creatures, and as such, we consume the majority of our information through images. The quality of an image has a direct bearing on its ability to convey information: to inform, to persuade, to entertain. Ray tracing remains a popular technique for generating high-quality images because it can produce a wide range of realistic visual phenomena, including accurate shadows, reflections and light-surface interactions. Since ray tracing simulates the physical travel of light through a scene, it can model complex lighting effects like caustics, multiple scattering in participating media and wavelength-specific behavior. Its physical basis makes ray tracing attractive both for general rendering and for scientific visualization, where physically accurate lighting can provide improved insight in addition to high image quality [37]. In addition, ray tracing can be used in radiative transfer simulations, where the end product is not an image but rather the physical model itself. Unfortunately, ray tracing's physical basis also makes it computationally more expensive than other image generation techniques, which limits its use in high-performance rendering. In particular, ray tracing has been considered too expensive for tasks that require real-time rendering, like video games, or for tasks that require operating on massive amounts of data, like large-scale visualizations or simulations.

Recent research [45, 91, 115] and improved hardware performance address ray tracing's computational bottleneck, but current hardware trends make efficient memory use increasingly important. At the chip level, the trend toward dense chip multiprocessors means that future chips will have less cache per core and more contention for bus bandwidth between DRAM and on-chip cache. At the large system level, the trend towards clusters and distributed file systems [103] means that future systems will have only a small amount of local memory per node and more contention for bandwidth between the file system to local DRAM.

The recursive ray tracing algorithm itself, however, remains fundamentally memory-inefficient. The unbounded memory requests caused by recursive ray tracing can create a memory bandwidth bottleneck, especially when rendering large scenes or when rendering complex lighting effects that use incoherent rays. Due to this bottleneck, users must choose between long render time or low render quality. Large data is often rendered only with simple lighting, when it can be rendered at all. Industrial users with dedicated machines might choose to spend additional hours or days rendering each image[1], but scientists with a limited allocation on a shared resource are typically unable to commit sufficient cycles for high-quality ray tracing. This cycle limit has an even greater impact on radiative transfer simulations for which simulation accuracy is critical: for example, large-scale cosmology simulations typically model cosmic radiation $1000\times$ more coarsely than they model gravitation due to the inefficiencies of the radiative transfer model [49].

[1]For example, each frame of a ray traced movie typically requires tens of hours: in 2006, *Cars* averaged 15 hours per frame; in 2009, *Avatar* averaged 30 – 50 hours per frame.

In this dissertation, we create a new interpretation of ray tracing that generalizes previous algorithms as points on a spectrum of *ray schedules* that organize ray and data operations. In particular, our dissertation contributes the following:

- it creates an analytic *memory performance model* for recursive ray tracing that exposes the contributing factors for the memory bandwidth bottleneck;
- it creates an algorithmic framework for dynamic ray scheduling;
- it presents a dynamic scheduling algorithm for a single processing core, targeting DRAM-to-L2 bandwidth, that achieves up to 100× savings in geometry bandwidth and 10× savings in total bandwidth;
- it presents a dynamic scheduling algorithm for distributed memory clusters, targeting disk-to-DRAM bandwidth, that achieves over 10× data load reduction and runtime improvement.

Through the development of our analytic model for recursive ray tracing, we show that repeated visits to dense regions of a dataset can increase the size of the working set, which is the data required to complete the computation. Once the working set grows beyond a particular size, it begins to evict itself from small, fast memory as it is processed. As a result, it will be loaded repeatedly into these upper levels of memory from the larger, slower memory below them. These repeated loads drive the consumption of memory resources during ray tracing and hurt runtime performance. By studying the memory utilization and the memory bandwidth

efficiency of a high-performance recursive ray tracer [91] under a range of rendering conditions, we corroborate the insights suggested by our model. We determine that when large models are ray traced with shading that uses incoherent rays, the order of memory requests can result in many loads from lower memory that create a memory bandwidth bottleneck, particularly when the working set exceeds available local memory. A ray tracing algorithm that can vary the order of memory requests can make ray tracing efficient for broader classes of visual effects across a broader range of data sizes.

Prior ray tracing algorithms lack the flexibility to effectively manage both rays and scene data. These algorithms each implement a static schedule for rendering operations, often one embedded in the implementation itself. Recursive ray tracers keep active ray state constant, usually at a single ray or a small collection of rays, while allowing active scene data to grow unbounded. In contrast, ray tracers that use computational reordering [85], which we call *queueing*, keep active scene data constant at each queue point while allowing active ray state to grow unbounded. Our broader formulation of ray scheduling shows these two algorithms to be points on a spectrum of ray schedules that actively optimize data movement and computation. We show that using queueing ray tracing, we can create algorithms that *dynamically schedule* groups of rays and scene data for efficient processing. Our approach schedules ray operations dynamically according to the data resident in targeted memory, whether the cache of a single processor or the main memory for each processor of a distributed-memory machine. By associating rays and data into self-contained groups with known memory bounds, our ray

tracer has the flexibility to schedule the processing of these groups to improve system performance. We show that dynamic scheduling of rays and data can improve performance for complex lighting algorithms and for large datasets where memory use limits performance, all without incurring significant communication overhead for smaller, compute-bound datasets. Our algorithm effectively manages data traffic in processor cache, often reducing the bandwidth consumption by an order of magnitude versus competing algorithms. Our algorithm can also render massive datasets on distributed-memory machines that competing algorithms cannot render. When rendering smaller datasets for which the other algorithms do complete, our dynamic scheduler significantly reduces traffic between local memory and disk. Our scheduler also exhibits better performance than static strategies for volumetric ray casting.

The remainder of the dissertation presents the supporting case for these contributions. We place our approach within the broad context of previous ray tracing work in Chapter 2; we compare our approach against specific related work in each chapter as appropriate. In Chapter 3, we motivate our work with an analytic evaluation of the bandwidth consumed by popular ray tracing algorithms. We confirm our evaluation in Chapter 4 with a study of the bandwidth performance of a state-of-the-art ray tracer. We present our dynamic scheduling algorithm and performance results targeting a single core machine in Chapter 5; and we present our dynamically scheduled distributed memory ray tracer and performance results targeting a large distributed-memory cluster in Chapter 6. We make concluding observations in Chapter 7.

Chapter 2

Background

This dissertation centers on the memory behavior of ray tracing algorithms. To understand this behavior, we must first understand how the ray tracing algorithm itself functions. In this chapter, we provide background on ray tracing and supporting concepts that will clarify our discussion through the remainder of this work. We give a ray tracing primer, including observations that underlie our new approach, in Section 2.1. In Section 2.2 we formally define key terms used in our later discussion.

2.1 Ray Tracing Fundamentals

Ray tracing is an image generation technique that simulates the travel of light through a dataset or scene to determine: first, which surfaces are visible from a particular viewpoint [9]; and then, how those surfaces are lit by the surrounding environment [119]. Since ray tracing is physically-based, it can be used to model a wide variety of realistic lighting effects and can generate photo-realistic images. High realism often comes with high computation cost, however, and simplifications are often made that reduce computational complexity while still generating a high-quality, if not photo-realistic, image.

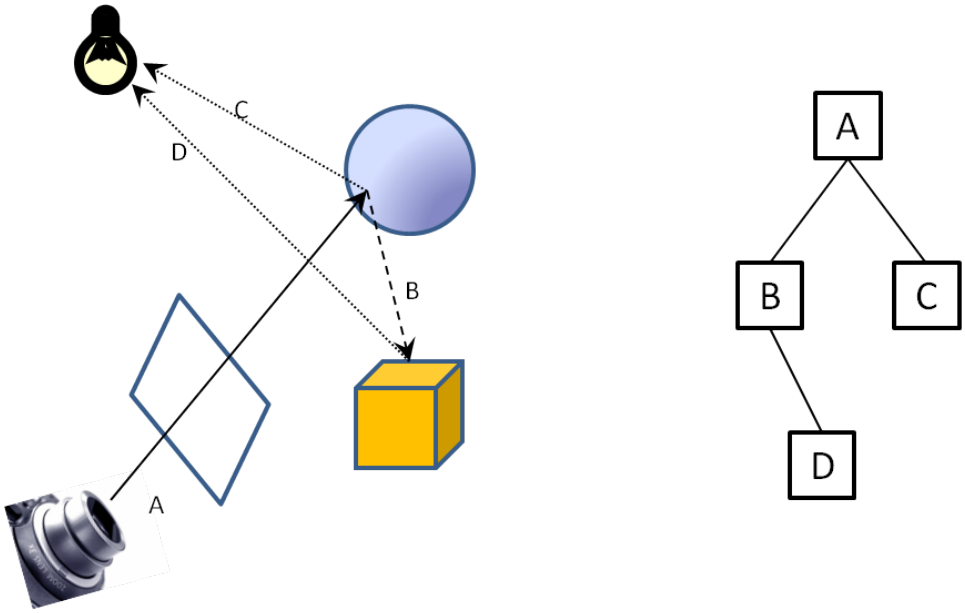

Figure 2.1: A ray tracing example. The left diagram shows a camera ray (A) traced into a scene and the secondary rays that are spawned (one reflection ray (B) and two shadow rays (C,D)); the right diagram contains the ray tree that corresponds to the rays on the left. A ray tracing algorithm determines in what order the rays in the tree are processed.

Ray tracing works by tracing simulated light rays though a scene, but opposite to the direction light actually travels. We are accustomed to light travelling forward, where light is generated by some emitter (the Sun, for example), certain wavelengths are reflected from objects around us, and some small percentage finally reaches our eye and registers as an image. Ray tracing models this process, but for efficiency, rays are traced "backward" from a viewpoint, often called the camera or eye, through an image plane, and into a scene (see Figure 2.1). These first rays that originate from the viewpoint are called *primary rays*, since they primarily determine what is visible from the viewpoint. It is common to model the viewpoint as a pinhole camera, which allows all primary rays to have the same origin. More complex camera models can be used [60, 87], but they require increased computational effort. Rays are usually generated with a *perspective* projection through the image plane, which makes rays fan out as they travel away from the viewpoint. Rays generated with an *orthographic* projection each travel parallel to the view direction, which can be used for rendering objects without perspective distortion.

Other rays generated during image generation are called *secondary rays*, since they calculate secondary effects that enhance the image generated by the primary rays. Any ray that is not a primary ray is called a secondary ray, regardless of whether it was generated from a primary ray or from another secondary ray. When a ray hits or *intersects* a surface, new secondary rays can be generated or *spawned*. Secondary rays model different aspects of light behavior: *shadow rays* sample whether an intersection point is visible to a light source; *reflection rays* model the reflection of light off a surface; *refraction rays* model the refraction of

light through a surface. Figure 2.1 contains both shadow rays (marked *s*) and reflection rays (marked *r*).

The rays generated in a scene can be represented hierarchically in a tree layout, called a *ray tree*, where each node in the tree corresponds to a ray generated during image generation. Each tree branch represents a "generates / generated-by" relationship: the node for each primary ray roots a tree that contains the nodes of all secondary rays generated by the primary ray and its descendents. We also use the term ray tree more generally to refer to the tree of all rays cast during rendering, which is formed by collecting the tree for each primary ray under a single root.

Consider the ray tree for a single primary ray. The root node, at level 0, represents the primary ray. The nodes at level 1, the primary ray's direct descendents, represent the secondary rays generated directly from the primary ray. The nodes at level 2 represent the secondary rays generated directly by the rays from level 1. Thus the nodes at level n of the tree represent rays generated by rays from level $n-1$ of the tree. See Figure 2.1 for an example ray tree.

At its core, a ray tracing algorithm does not determine *which* rays are put into the ray tree (that is a function of the lighting and shading used in the scene), but rather *in what order* the rays in the ray tree are processed. Most ray tracers fundamentally operate according to Whitted's recursive algorithm [119], called *Whitted-style* or *recursive* ray tracing[1]. Figure 2.2 contains pseudocode for this recursive algorithm. The algorithm results in a depth-first traversal of the ray tree for each

[1]Note that path tracing [56] operates recursively, but it generates only one ray at each intersection rather than the tree of child rays common in a Whitted-style tracer

```
function MakeImage(pixels)
{
  i = 0;
  primary_rays = GeneratePrimaryRays();
  foreach ray in primary_rays
  {
    pixels[i] = TraceRay( ray );
    i = i + 1;
  }
}

function TraceRay(ray)
{
  closest_obj  = NONE;
  closest_dist = MAX;
  pixel_color  = Blank();
  foreach object in scene
  {
    if (Intersects( ray, object ))
    {
      if (Distance( ray, object) < closest_dist)
      {
        closest_obj = object;
        closest_dist = Distance( ray, object );
      }
    }
  }

  if (closest_obj != NONE)
  {
    secondary_rays = GenerateSecondaryRays( ray, object);
    foreach r in secondary_rays
    {
      pixel_color = pixel_color + TraceRay(r);
    }
  }
  return pixel_color;
}
```

Figure 2.2: Pseudocode for Whitted's recursive ray tracing algorithm.

primary ray. Thus, once tracing begins on a primary ray, the algorithm will trace all descendent rays of that primary ray before beginning another primary ray. We will explore the implications of this ray ordering in Section 2.1.3.

2.1.1 Acceleration Structures

Each ray traced solves an instance of the *visibility problem*: what can be seen in the ray's direction from the ray's origin? Generally, two outcomes are possible: the ray intersects an object, or the ray exists the scene. However, multiple objects may lie along a ray's path, and the tracer usually must determine *the nearest* object a ray intersects[2]. Early ray tracers performed poorly on complex scenes, scenes that contain many objects, because they solved the visibility problem for each ray by testing it for intersection against every scene object [102]. All these ray-object intersection tests were necessary because, other than the objects themselves, there was no spatial information in the scene description that could be used to determine whether a ray might hit a particular object.

Acceleration structures were created to provide additional spatial information to the ray tracer. These structures reduce ray-object intersection tests by partitioning, and thereby sorting, scene space. Objects not in a ray's path are quickly eliminated from further consideration and only objects that are likely to be intersected are actually tested. *Traversal* of a ray through an acceleration structure proceeds as follows. A ray is first intersected against the acceleration structure, and

[2]Shadow rays are a notable exception to this condition. It is common to terminate shadow ray traversal when any intersection is detected, since one occluding object is enough to block the light source for which the shadow ray is testing visibility.

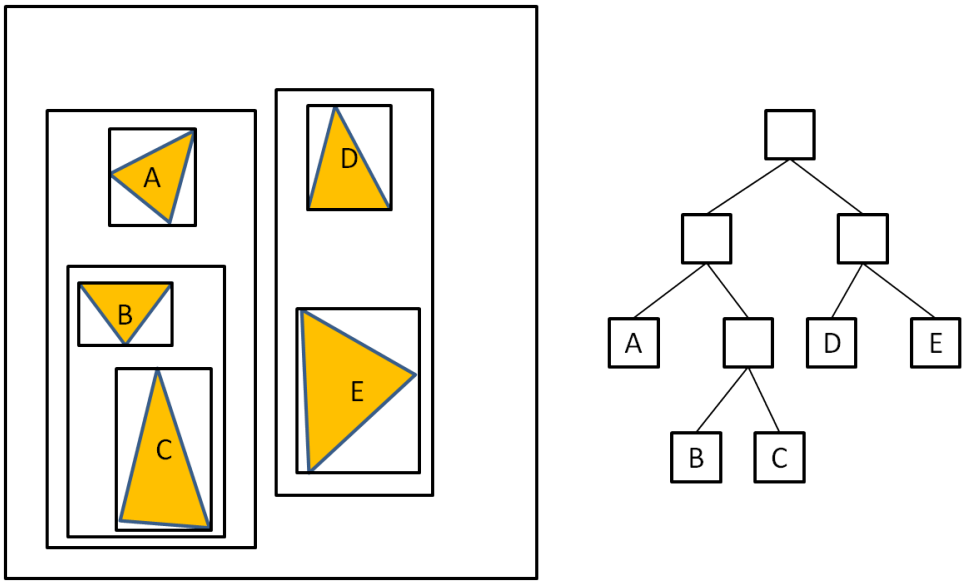

Figure 2.3: Bounding volume examples, projected into 2D. Bounding volumes contain each triangle (left). Outer boxes form a bounding volume hierarchy (BVH). Since they subdivide object space, the boxes can overlap. A tree representation of the bounding volume hierarchy (right) can be used to determine which bounding volumes must be considered at each level. A BVH limits bounding volume tests since sub-volumes need not be tested if a ray misses an enclosing volume. The BVH here uses a branching factor of two, but higher branching factors can be used. The hierarchy boxes are normally tight-fitting, but they have been expanded here for clarity.

only the intersected parts of the structure are considered further. The intersected parts are usually ordered by increasing distance along the ray, and the ray is tested against the objects contained in each intersected part until the nearest object intersection is found.

Acceleration structure partitions can be created either in *object space* or in *scene space*. An object space partition forms non-intersecting sets of objects, each represented by a bounding volume [92] that tightly encloses the space occupied by

the object set. Rays are tested for intersection against these bounding volumes first, and if the ray misses the bounding volume, the object is not tested. This reduces the number of object tests, which are generally more expensive than bounding volume tests. However, because the bounding volumes do not subdivide scene space, a ray must be tested for intersection against all bounding volumes to find the closest intersection. A hierarchy of bounding volumes (see Figure 2.3), where increasingly larger volumes tightly enclose several smaller volumes, helps limit the number of bounding volume tests at each level of the hierarchy. Intersection tests are then performed only on objects within the bounding volumes hit by a ray. If these bounding volumes are considered in order of increasing distance along the ray, more distant volumes and their objects can be discarded if a closer object intersection is found. Bounding volumes may overlap in scene space, so only volumes that do not overlap the potential object intersection point can be discarded; the objects in bounding volumes that overlap the point must also be tested for intersection, in case a closer intersection can be found.

In contrast to an object space partition, a scene space partition forms distinct spatial regions. These distinct spatial regions are usually formed by linear divisions, such as with a grid [29] or with planes [33], and they eliminate the extra intersection tests necessary when object space partitions overlap. Rays are first tested for intersection against the spatial regions, and because the regions do not overlap, an intersection point found in one region is guaranteed to be closer than any possible intersection point in a more distant region. Thus, any objects contained in those distant regions can be rejected without explicitly testing them for

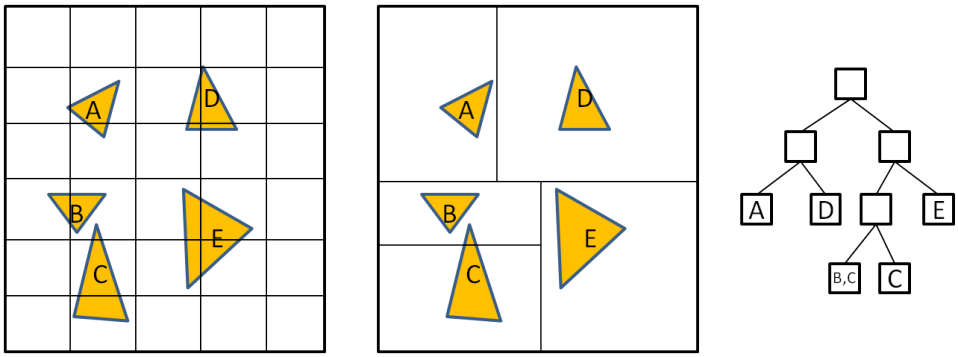

Figure 2.4: Space partitioning examples. A regular grid (left) divides space evenly, but it does not adapt to dense regions of data. A k-d tree (center, right) adapts by subdividing the dense region with additional planes. Objects that overlap space partition boundaries must be addressed by the construction algorithm. Typically, such objects are either divided, so that each part fits into a single region, or duplicated, so that a copy is put into each region. Here, object C has been duplicated. Note that the k-d tree planes are usually tight-fitting, but they have been spaced here for clarity.

intersection. A regular scene space partition, one that creates evenly-sized spatial volumes, can create inefficiencies. It might subdivide space too coarsely, so that there are many objects in each region, or too finely, so that there are many empty regions to traverse. An *adaptive* space partitioning algorithm attempts to balance the number of objects in each region and thereby limit the number of intersection tests required, either by adjusting the division boundaries or by subdividing regions that contain too many objects. Adaptive algorithms often create a spatial hierarchy by subdividing only regions with many objects; regions with few (or no) objects remain large to limit traversal operations. Adaptive algorithms usually select partitions heuristically to balance structure quality against construction time. Figure 2.4 shows both a regular and an adaptive scene space partition.

Our work focuses on the behavior of acceleration structures that partition scene space. In particular, our work uses *k-d trees* [34], an adaptive algorithm that recursively subdivides scene space with axis-aligned planes according to a subdivision heuristic. These subdivisions are represented by a binary tree where each internal node represents a subdivision plane, the branches that descend from each internal node represent the two subregions created by the node's plane, and each leaf represents a subregion containing objects (see Figure 2.4). Branches that represent empty subregions can be pruned, which makes traversal of the tree more efficient.

The traversal of a k-d tree proceeds as follows. First, an initial test is made against a bounding volume that contains the entire scene, to eliminate any rays that miss the scene entirely. If the ray hits the scene, the ray is tested for intersection against the plane represented by the root node of the k-d tree. Three outcomes are possible:

- the ray travels only in the subregion represented by the left branch of the tree, so the plane on the left branch is considered next and the right branch is discarded
- the ray travels only in the subregion represented by the right branch of the tree, so the plane on the right branch is considered next and the left branch is discarded
- the ray intersects the plane and both branches are kept, with the plane on the closer branch considered first and the plane on the distant branch pushed on

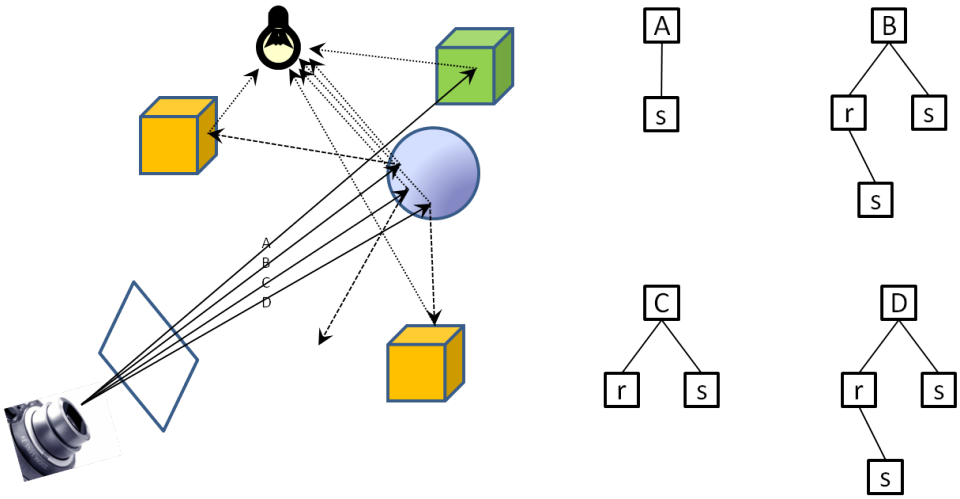

Figure 2.5: This diagram shows how single-ray and packet-based recursive algorithms traverse rays. A single-ray traversal completes all the rays in one tree before beginning the next: both the primary and shadow ray in tree A, then the four rays in tree B, the three rays in tree C, and last the four in tree D. A packet traversal traces the primary rays for A, B, C and D together in one packet. Depending on the implementation, the three shadow rays generated off the sphere may be traced together, but the reflection rays and the shadow rays off each cube are likely traced individually. Note how the primary rays tend to remain coherent as they travel through space, but the reflection rays spawned by those primary rays quickly lose coherence.

a stack for later consideration.

This decision is repeated for each plane until a leaf is reached and ray-object intersections are performed against the leaf's objects. If no object intersection is found, a plane is taken from the top of the stack and traversal resumes. This continues until an intersection is found or until the stack is empty, which means the ray missed all objects in the scene.

2.1.2 Selecting Rays for Traversal

Recursive ray tracing often traces a single ray at a time, as presented in Figure 2.2. For rays that traverse similar paths through the acceleration structure, however, it can be more efficient to trace these rays together. We will call rays *coherent* if they traverse the same part of the acceleration structure at the same time. Coherent rays can be traced together, represented either as a continuous beam [5, 43] (where member rays are not explicitly defined) or as *packets* of discrete rays [115]. The wide SIMD instruction support available on modern processors has helped make packet tracing popular in recent ray tracers [16, 18, 30, 91, 105]. Because of its popularity, we include packet tracing as a test case in many of our experiments.

Algorithmically, packet tracing operates exactly like recursive ray tracing but with each traversal and intersection operation applied to multiple rays at once. If the rays do not share identical direction vectors, the rays will become less coherent as the packet moves through scene space, and some traversal and intersection steps will not apply to every ray. When this occurs, a mask is applied to the packet so only the proper rays are affected by each operation. As rays in the packet lose coherency, there is less benefit realized by processing them together. When packet coherence is lost, a packet tracer will usually fall back to single-ray recursive tracing. Figure 2.5 shows ray selection for both single-ray and packet-based recursive tracing.

2.1.3 Ray Scheduling

A *ray schedule* defines the order in which rays in a ray tree are traversed. Every ray tracer employs a schedule for its work, though a recursive tracer defines

the schedule implicitly, and therefore statically, within its algorithm. A recursive tracer schedules rays according to a formal depth-first or breadth-first traversal of each ray tree, though as mentioned in Section 2.1.2, several trees can be traced simultaneously using a packet or a beam.

Recently, two new classes of ray tracers have been developed that can schedule rays differently from a recursive tracer: *reordering* ray tracers [18] that can change the order in which child nodes are visited on each ray tree branch, but that do not pause rays during traversal; and *queueing* ray tracers [3, 19, 25, 76, 85, 99] that can both reorder rays and enqueue them at points during acceleration structure traversal, effectively pausing their traversal until a later time when all rays in a selected queue are traversed together.

We contend that these seemingly disparate ray tracing algorithms actually implement particular points on a spectrum of ray scheduling algorithms, where the schedule defines the rays that are traced at each algorithm step and over which data the selected rays are traced. In this section, we compare these three classes of ray tracers and how the schedule each uses can affect rendering performance.

2.1.3.1 Non-reordering Recursive Ray Tracers

Most ray tracers use a non-reordering recursive approach based on Whitted's algorithm [119], where a ray is traced to completion once the tracing of it has begun. These tracers use an implicit, static ray schedule: operate first on a set of camera rays, then trace each child ray (alone or in sets) before tracing another set of camera rays. Put another way, this ray schedule performs a depth-first traversal

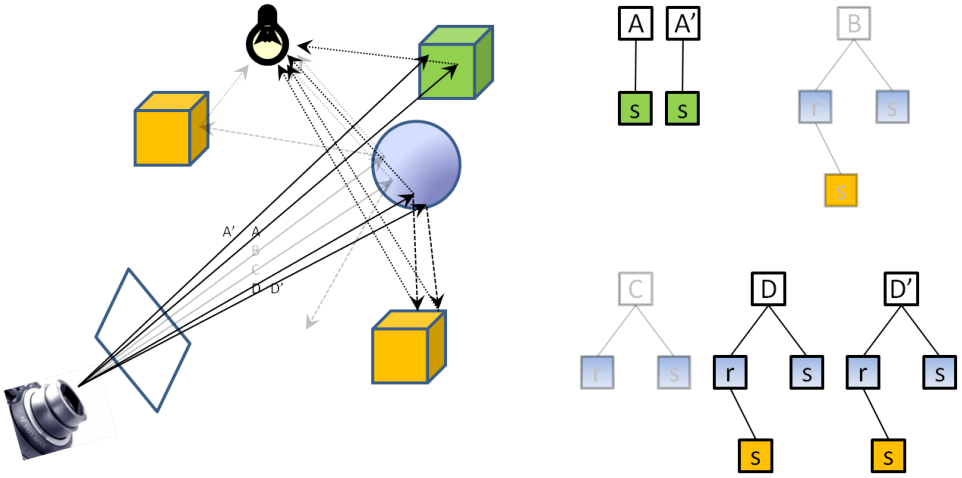

Figure 2.6: This diagram shows a traversal order of the scene from Figure 2.5 when rendered with a reordering tracer. The initial traversal proceeds much like the packet traversal example, where primary rays A, B, C and D are traced in one packet. However, the ray trees are traversed breadth-first, so while primary rays are traced in packets, there is a reordering step before any child ray is traced. The child rays can be regrouped into new packets. Here, the rays spawned from A might be grouped with rays spawned by the previous primary packet (A′), and rays spawned from D might be grouped with rays spawned from the subsequent primary packet (D′).

of each primary ray's tree. As a result, the schedule is set by the size of each set of camera rays and the order in which the camera ray sets are traced (see Figure 2.5).

Because the ray schedule is set at the start of rendering, it cannot be adjusted to build ray coherence or to react if the working set grows large in memory. If the entire working set cannot fit in memory, this rigid schedule can cause thrashing as rays re-request evicted data. Thrashing can be especially severe for incoherent rays, since their processing can access disparate regions of scene space and thus increase the size of the working set.

2.1.3.2 Ray Reordering Tracers

Several recent ray tracers [17, 18] augment the classic recursive algorithm by modifying ray sets during traversal and between generations of child rays. In particular, these tracers reorder rays into more coherent sub-groups within a ray set as they are traversed, and they can re-form ray sets between generations. Note, however, that the ray reordering occurs only within the original ray set, and the ray set itself remains intact until all member rays are successfully intersected. Since ray sets are re-formed between ray generations, the schedule is determined by the set of rays formed at each generation. Figure 2.6 contains an example.

This more flexible approach can build ray coherence by assembling sets of rays with similar origins and directions; it also makes efficient requests to memory by assembling ray sets that use the same shader and by reordering rays to improve acceleration structure efficiency. Tracers in this class, however, do not enqueue rays in the acceleration structure as the rays are traversed. Once the traversal of a ray set begins, it will be completed for all rays in the set: no ray is ever deferred or "paused" during its traversal. Therefore, these reordering tracers can still thrash memory if the entire work unit does not fit in core.

2.1.3.3 Queueing Ray Tracers

Separately, researchers are exploring queueing[3] ray tracers that enqueue rays in regions of data space to build working sets with many rays and a known,

[3]Pharr et al. refer to this algorithm as "computation reordering". We coin the term "queueing" to better differentiate these tracers from those that only reorder rays

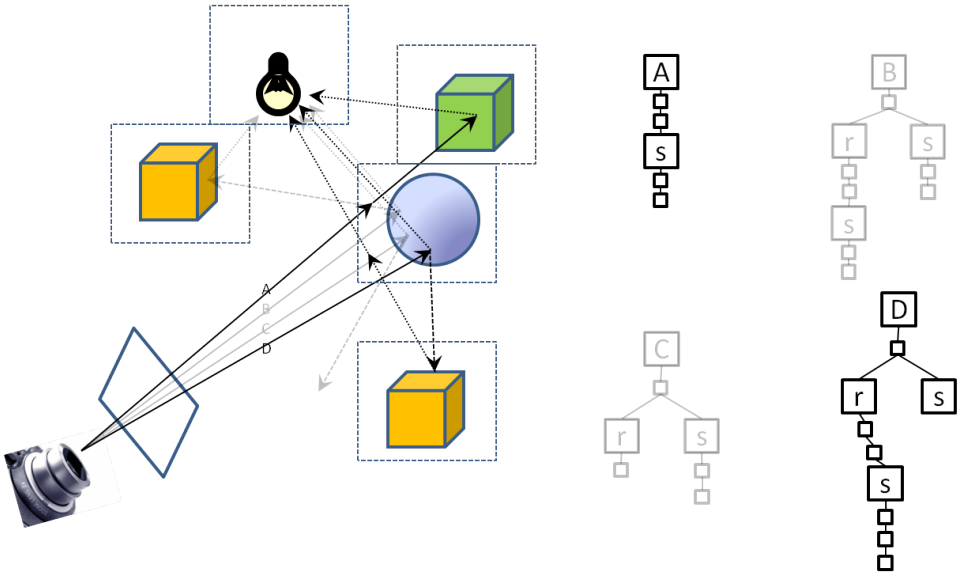

Figure 2.7: This diagram shows a traversal order of the scene from Figures 2.5 and 2.6 when rendered with a queueing tracer. The queue points are marked in the scene by stippled boxes and in each ray tree by small nodes. The flight of a single ray can be divided into multiple steps through the scene, with other computation occurring between steps. For example, the primary ray in tree A and the shadow ray in tree D are each enqueued first at the queue point around the sphere. After they are tested for intersection, they continue to their next queue point. Note that the primary ray and second shadow ray for tree D cannot be in the sphere's queue at the same time because there is a direct dependency between them: the primary ray must intersect and generate the reflection ray, which must then intersect before the shadow ray can be generated.

bounded quantity of data [3, 19, 25, 76, 85, 99]. For tracers in this class, rays are traced iteratively rather than recursively. As a result, the traversal of a ray can be deferred or "paused" at each queue point and resumed at a later time, which allows the traversal computations to be rearranged to achieve system performance goals. The ability to defer ray traversals distinguishes these tracers from the previous classes described.

Each ray queue forms a dynamic ray set that can exploit coherence for rays without requiring that they share a common origin, a similar direction, or even the same generation in the ray tree. Exploiting this broader notion of ray coherence requires additional state per ray and increases computational complexity. The schedule for these tracers is determined by the order in which queues are selected for processing. See figure 2.7 for an example. Tracing rays in this way enables the dynamic scheduling of ray queues that lie at the core of our algorithms.

Further, because each queue point builds a dense collection of *both* ray and scene data, we observe that the operations over the rays at a queue resemble regular loop iterations used in matrix computations. As such, we can now apply system-level loop optimizations for regular data [7, 20, 69, 73, 120] that are not normally applicable to the irregular computation structure of recursive ray tracing. We believe that we are the first to make this observation.

2.2 Formal Definitions

We now provide formal definitions to concepts central in our discussion. Some of these terms have already been introduced informally in this chapter. These

definitions will help distinguish nuances among the various algorithms we discuss and among the various forms of coherence these algorithms exploit to make efficient use of the memory system.

2.2.1 Algorithmic Definitions

ray — a discrete approximation of one segment of light travel through a region of space. We consider a ray's lifetime to begin when it is spawned and to end when it is successfully intersected. Any rays generated as a result of the intersection (reflection rays, refraction rays, shadow rays) are considered distinct descendants on the ray tree. An intersection with a semi-transparent surface generates a new ray, so that we do not consider multiple intersections or bounces for a single ray. In the special case of volume ray casting, a ray's lifetime runs from when it is spawned until when it accumulates maximum opacity.

ray tree — a tree that represents the parent-child relationships for a single primary and its descendants, or for all rays generated during the rendering of an image. Each node in the tree represents a single ray, and each branch represents the relationship between the intersection of a ray and the spawning of each of its descendants.

ray operations — the calculations required to successfully trace a ray. For surface rendering, ray operations include acceleration structure traversal and object intersection tests. For direct volume ray casting, ray operations include traversing the dataset and sampling the data at intervals along the ray path.

atomic execution — the complete set of ray operations that must occur together for a given ray under a particular algorithm. For recursive tracers, the atomic execution period of a ray covers all ray operations performed on it. For queueing tracers, however, an atomic execution period covers only the work done on a ray for a single queue, and total operations on a ray will be divided into an atomic execution period for each queue the ray enters.

ray set — a collection of rays that are processed together in a single atomic execution period. The set may contain a single ray; a discrete collection of rays (a ray packet or a ray queue); or a continuous collection of rays (a ray beam or cone).

unit of work — (or **work unit**) a ray set and the data needed to process one atomic execution period for it. For a recursive tracer, each ray belongs to exactly one unit of work that covers the lifetime of each ray it contains. For a queueing tracer, each unit of work covers only the time a ray is in a particular queue, and a ray may be part of several work units during its lifetime.

schedule — an ordering imposed on a series of work units. A schedule can be represented by the traversal order of branches on the ray tree.

2.2.2 Rendering Coherence Definitions

Sutherland, Sproull and Schumacker define *coherence* in graphics as "the extent to which the environment or the picture of it is locally constant" [102]. Later, Green and Paddon [35] defined coherence terms specifically for ray tracing effi-

ciency. We will refer to Green and Paddon's terms, defined below, throughout the remainder of this work to distinguish the memory behavior of ray tracing algorithms and to explain their performance.

image coherence — rays traced through adjacent pixels are likely to travel through the same regions of scene space, traverse the same acceleration structure nodes, and intersect the same object. Tracing primary rays that pass through contiguous pixels, such as in tiles [115] or along a space-filling curve [104], exploits image coherence. Tracing primary rays from widely separated pixels is image-incoherent, but can achieve good load-balancing for parallel recursive ray tracers [36, 114].

ray coherence — rays that travel a similar path are likely to require the same data for their traversal and intersection computations. Ray packeting [115], and similar techniques that trace a group of rays together, exploit ray coherence to achieve better performance. Pharr et al. [85] expand this concept to include rays that occupy the same region of scene space simultaneously, regardless of their origins or directions. We use this broader definition in our work. Secondary rays generated by ray-coherent primary rays often do not remain ray-coherent themselves; ray reordering [18] and ray queueing [3, 19, 25, 76, 85, 99] techniques build ray-coherent groups of secondary rays.

data coherence — objects that are nearby in scene space are stored in nearby locations in machine memory. This relationship translates the coherent references of an algorithm into coherent requests in memory. Data coherence must exist

for image- or ray-coherent traversals to maximize efficient use the memory system [35, 78, 85]. Otherwise, coherent accesses in the scene might result in random requests in memory, eliminating the coherent benefit.

Chapter 3

Modeling Bandwidth Consumption

This thesis approaches ray tracing from a systems perspective, so we first examine the impact of recent hardware trends on ray tracing algorithms. Modern chips with many processing cores promise to provide ample processing power for high-performance ray tracing calculations, and ray tracing's embarrassingly parallel nature would seem to lend itself well to such architectures. However, these multi-core architectures introduce a new bottleneck in the memory-system, because bandwidth to the lowest-level of cache must be shared among many cores. This contention can be exacerbated when ray tracing with a complex lighting model, which is necessary for photo-realistic images. Complex lighting algorithms, such as Monte-Carlo methods and photon mapping [53], can generate incoherent memory accesses and can require the use of additional data structures [53]. We conclude, then, that improving the memory efficiency of ray tracing will facilitate high-performance ray tracing of scenes with complex lighting.

We believe recursive ray tracers [119] are not memory-efficient because they traverse rays depth-first. In a depth-first traversal, consecutive primary rays might be tested for intersection against the same geometry, but these tests can be widely separated in time: 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 can be masked, but that becomes less likely as rendering trends point to larger scenes rendered using more realistic lighting models. Optimizations such as the tracing of rays in SIMD-friendly packets [115] or the use of ray frustums [91] help, but only if rays are sufficiently coherent, which is typically only the case for primary and perhaps shadow rays. Unfortunately, they offer little benefit for incoherent rays in a realistically-lit scene.

In this chapter, we evaluate the memory behavior of recursive ray tracing through an analytic model of it bandwidth consumption. Our model exposes the key factors that impact the algorithm's memory behavior and enables us to understand the mechanisms that cause its memory inefficiencies, particularly for incoherent rays. This insight will be instrumental in the design of our new algorithm presented in Chapter 5.

3.1 Overview

In this chapter, we introduce an analytic model for describing ray tracing systems that can reasonably predict bandwidth consumption. We choose bandwidth consumption as our performance metric because we believe bandwidth to be the primary bottleneck for future ray tracing systems. The model presented can be adapted to measure computation by replacing the memory-specific terms with instruction-specific counterparts, but a thorough treatment of a computation model is beyond our scope here.

By exposing the various factors of ray tracing that contribute to bandwidth

consumption, our model allows us to make educated choices about system design. Without loss of generality, we limit our discussion to the k-d tree acceleration structure [34], since it is supported by many current high-performance ray tracers [16, 91, 115].

After a discussion of related work in Section 3.2, we explain the operation of a computer's memory system and the importance of bandwidth efficiency in Section 3.3. We present our model in Section 3.4 and we evaluate its predictive capability against the measured bandwidth consumption of an experimental ray tracer in Section 3.5.

3.2 Related Work

Several recent research projects target ray tracing performance issues. These efforts focus primarily on practical acceleration structure efficiency, whereas we investigate the algorithmic effects of the traversal upon the acceleration structure.

Vlastimil Havran's dissertation [39] provides an excellent study on the empirical performance of many ray tracing acceleration structures. Hurley et al. [47], describe an analytic model for building efficient k-d trees. Warren Hunt's dissertation [46] contains the current best practices regarding fast construction of efficient k-d trees.

The turnover terms of our model are inspired by the intrinsic and extrinsic interference terms in the general cache model of Agarwal et al. [2]. Our initial model of turnover was inspired by the cache model designed by Lam et al. [63].

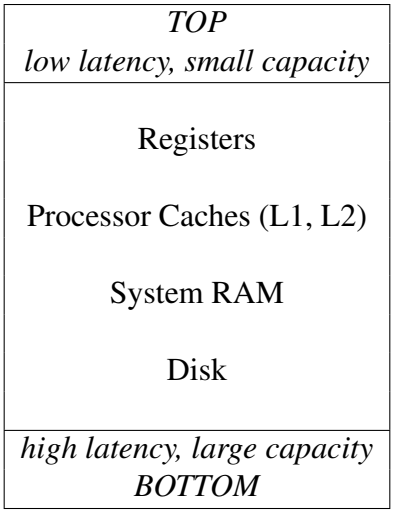

Figure 3.1: The memory system hierarchy for a typical machine.

3.3 The Memory System and Why Bandwidth Matters

A computer's memory system fetches data required by the computational units (the ALUs) of a processor and stores the results of computation for later use. The memory system is organized as a hierarchy of data storage devices to help ensure that data is supplied quickly when requested, as shown in Figure 3.1. Memory devices that are close to the computational units can fetch stored data quickly (they have low *latency*), but these devices cannot store much data (they have small *capacity*). Memory devices that are further away from the computational units have higher latencies but also larger capacities. Each level of the memory hierarchy acts as a *cache*, in that data stored at its level does not need to be retrieved from a lower, slower level of the hierarchy. For example, memory requests to system RAM typically have much higher latencies than do requests to processor cache, so data kept

in processor cache can be supplied more quickly to the computational units of the processor.

Bandwidth expresses the rate at which data can be moved between two locations. In our context, it is the rate at which data can be moved between adjacent levels of the memory hierarchy. Specifically, we are interested in read bandwidth: the rate at which a level of memory can supply data to the next higher level. Data can move no faster than this rate, but it can move much slower if the amount of data requested exceeds the available bandwidth. In this case, bandwidth becomes *saturated* and excess data must wait until the preceding data have been moved. Thus, too many data requests, too much *bandwidth consumption*, can saturate bandwidth and delay the entire computation.

We will construct our model to estimate the actual amount of data requested *and* to identify the particular algorithm behavior that generates the requests. In so doing, we can identify the aspects of the tracing algorithm that should be modified or replaced if a bandwidth problem exists.

3.3.1 "Coherency", "Locality" and the Memory System

Graphics literature and computer systems literature have adopted conflicting terminology for similar memory concepts. In graphics, *coherency* refers to repeated accesses to the same data, like the same object in a scene. In systems, this concept is closest to memory *locality*, where data that reside nearby in address space tend to move together through the memory hierarchy (since loads move blocks of data, not individual addresses) and so requests for nearby data tend to succeed without

an additional load. Systems literature defines *coherency* as data consistency across multiple memories, either between levels of the memory hierarchy or between separate memories at the same level. In our discussion, we use coherency and coherent as defined for graphics to minimize confusion with related graphics work.

Coherent (*spatially local*, in systems terms) memory requests use the memory system efficiently because data stays high in the hierarchy where it can be accessed quickly. An incoherent request can replace the data in the upper memory hierarchy and if that data is needed later, it must be loaded again. These data reloads increase bandwidth consumption and too many will saturate bandwidth.

We want our model to express the coherency of memory requests, so we can determine how much data will be reloaded. Below, we define several terms for memory system coherency that will influence our model's design.

memory coherent ray access — two rays r_1 and r_2 are coherent with respect to memory access if they traverse the same path through the acceleration structure and all rays that are processed between the time r_1 is processed and the time r_2 is processed also traverse that same path.

memory-coherent ray set (MCRS) — a maximal set of coherent rays is called a coherent ray set (MCRS): i.e., neither the ray before nor the ray after the rays in the MCRS traverse the same path as the rays in the MCRS. This should not be confused with a **ray set** as defined in Section 2.2, where the set is defined by the processing order of the algorithm. In contrast, a MCRS is defined by the path the member rays take through the acceleration structure. An MCRS

may span several ray sets, or a ray set could contain several MCRSs. A MCRS may also contain subsets of several ray sets. See Figure 3.4.4 for an example of memory coherent ray sets.

memory-coherent data set (MCDS) — the data loaded in the process of traversing a memory-coherent ray set. This is similar to a **unit of work** as defined in Section 2.2 applied to a memory-coherent ray set. Whereas MCRS describes a logical set of rays, MCDS refers to the physical data loaded, which includes ray, node and geometry data.

memory-coherent set (MCS) — an MCRS and its corresponding MCDS.

head leaf — the starting leaf of an MCRS. For the special case where the ray starts outside the acceleration structure (some camera or shadow rays, for example), the root acts as the head leaf.

tail leaf — the ending leaf of a coherent ray set. A leaf may be the tail leaf for multiple MCRSs.

3.4 The Bandwidth Consumption Model

In this section, we present our model as equations that estimate the bandwidth consumed by a recursive ray tracer. We attempt to keep the discussion of acceleration structure as general as possible, but we assume a k-d tree at points that call for specificity. The terminology we use reflects this assumption (i.e., node, leaf). However, we use *node* to mean any part of the acceleration structure, and *leaf* to mean specifically a terminal unit at which geometry must be intersected.

We have identified several bandwidth **consumption factors** that affect the total bandwidth consumed during rendering, which we discuss in Section 3.4.1. These factors interact through several **consumption modes** of the rendering process that we describe in Section 3.4.2. We then derive the consumption equation itself in Sections 3.4.3 and 3.4.4.

3.4.1 Bandwidth Consumption Factors

The bandwidth consumption factors are initial conditions for the ray tracing algorithm, which can be determined before ray traversal begins. These factors can be aspects of the ray tracer implementation (packet size, ray traversal order), or they can be supplied by inputs (frame buffer resolution, geometry distribution). Each factor is represented in our consumption equation by a parameter or a combination of parameters, which are described in Table 3.1.

framebuffer resolution and sampling rate — the resolution of the framebuffer combined with the rate at which it is sampled (i.e., number and location of rays per pixel) determine both how many primary rays will be traced and the spacing between these rays.

camera location and view direction — the location of the camera and the direction it points both determine which part of the scene and acceleration structure will be accessed by primary rays. Shadow rays and other secondary rays may access other parts of the scene, but even these are indirectly influenced by the original camera position and view direction.

ray traversal order — the order in which primary rays are selected to be traced through the acceleration structure. This order determines when particular parts of the acceleration structure and scene geometry will be accessed.

packet size — the number of rays traced simultaneously. The origins and directions of the rays determines the packet's ray coherence (see Section 2.2.2). Larger packets allow more rays to be traced simultaneously, but they typically lose ray coherence more quickly than smaller packets.

geometry distribution — the distribution of geometry through the scene affects the shape and quality of the acceleration structure that is produced, which in turn affects which geometry is accessed at each leaf.

acceleration structure adaptiveness — the adaptiveness of the acceleration structure determines how well the structure can separate regions with high concentrations of geometry into small groups. Note that achieving smaller groups of geometry increases the depth of the acceleration structure and the number of nodes that must be loaded for traversal.

3.4.2 Bandwidth Consumption Modes

The consumption factors described above interact through four **consumption modes** of the rendering algorithm: compulsory, average turnover, large-leaf turnover and saturation-leaf turnover. Each mode is represented by a term in the bandwidth consumption equation, which are described in Table 3.2.

compulsory consumption — the total cost of each ray, node and geometric object

loaded during rendering. Each ray traversed, and each node and object tested for intersection, must be brought into cache at least once.

average turnover consumption — the cost of reloading ray, node or object data that has been loaded previously but is no longer in cache.

large-leaf turnover consumption — the cost of intersecting rays against a leaf that contains enough geometry to occupy a significant percentage of the available cache (a **large leaf**). Processing such a leaf causes the displacement of data that would otherwise be accessed coherently under average turnover.

saturation-leaf turnover consumption — the cost of intersecting rays against a leaf of the acceleration structure that contains more data than can fit in available cache. When such a leaf is processed, all previously cached data is evicted from cache. Any coherence from caching is lost, and the leaf data must be reloaded for each ray, or SIMD ray group, in a large ray set. These saturation leaves can occur in geometrically-dense regions of the scene where the acceleration structure algorithm failed to sufficiently separate the objects, whether due to an algorithm deficiency, a depth-vs-quality heuristic, a time-vs-quality heuristic, or some combination of reasons.

3.4.3 General Consumption Equation

The bandwidth required for a rendering pass (BW) is composed of (1) the bandwidth to load the rays traced (BW_r) plus (2) the bandwidth to load nodes of the acceleration structure for ray traversal (BW_n) plus (3) the bandwidth to load

$COST_r$	cost (in bytes) to load a ray
$COST_n$	cost (in bytes) to load a node of the acceleration structure
$COST_g$	cost (in bytes) to load a unit of geometry
R	total number of rays traced
R_a	the number of rays traced together (e.g., the ray set size)
p	the number of rays able to be processed in parallel (e.g., via SIMD processing)
n	expected number of acceleration structure nodes loaded per coherent ray set
l	expected number of acceleration structure leaves loaded per ray. Note that $l \leq n$
g	expected amount of geometry contained within a leaf node.
c_n	number of *coherent ray accesses* per internal node.
c_l	number of *coherent ray accesses* per leaf node. In the equations below, we will use c_l as a conservative estimate for c_n.

Table 3.1: Parameters for Equations 3.1–3.9.

BW	total bandwidth consumed, i.e. the number of bytes required to complete rendering operations
BW_r	bandwidth consumed for ray data
BW_n	bandwidth consumed for acceleration structure node data
BW_g	bandwidth consumed for geometry data
C	compulsory bandwidth consumption, which is exactly the bandwidth cost of the first touch to all touched nodes and geometry.
θ_a	*average turnover* of data in the cache; the amount of data expected to be reloaded into the cache as a result of incoherent access to scene data.
θ_l	*large-leaf turnover* of data in the cache; the bandwidth consumed by processing a leaf that is larger than the expected leaf size, but not large enough to fill the entire cache[1].
θ_s	*saturation-leaf turnover* of data in the cache: the bandwidth consumed in processing a leaf that contains more geometry than can be cached at once[2].

Table 3.2: Derived Terms for Equations 3.1–3.9.

geometry for intersection testing (BW_g). We can express bandwidth as the sum of these three terms:

$$BW = BW_r + BW_n + BW_g \tag{3.1}$$

This is the most general formulation of the bandwidth consumption equation. Reading the equation from left-to-right, it implies a causal relationship among the sources of bandwidth consumption: rays are traced through some number of nodes. Some of these nodes (the leaves) contain geometry, and those objects must be loaded for intersection. Reading right-to-left presents the negative causal relationship: geometry is not loaded unless the node that contains it is loaded. A node is not loaded unless a ray pierces it.

Unfortunately, this form does not distinguish between data that must be loaded for the computation and data that is reloaded due to incoherent accesses. Therefore, we reformulate the bandwidth equation as the sum of terms that represent each of the bandwidth consumption modes described in Section 3.4.2: compulsory, average turnover, large-leaf turnover, and saturation-leaf turnover. These last three terms model bandwidth consumption by leaves that contain increasing amounts of geometry: those smaller than and up to the mean size, those between the mean size and the cache size, and those larger than the cache size (see Figure 3.4.3). We will find in our subsequent analysis that large- and saturation-leaves exacerbate incoherent accesses because they replace large amounts of cached data.

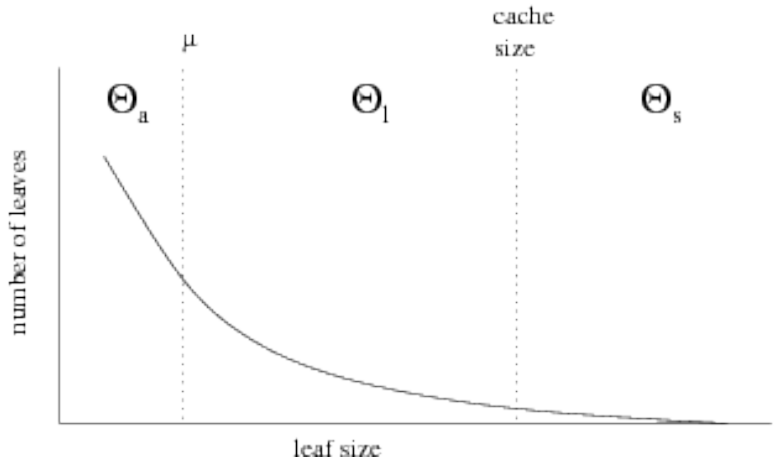

Figure 3.2: Terms as Related to Leaf Size — the terms average turnover (θ_a), large-leaf turnover (θ_l), and saturation-leaf turnover (θ_s) account for the span of possible leaf sizes. θ_a represents leaves up to the expectation μ. θ_l represents leaves that are larger than the expectation but smaller than the cache size. θ_s represents leaves that are larger than the cache.

$$BW = C + \theta_a + \theta_l + \theta_s \tag{3.2}$$

While these general formulations confirm our basic insights, we want a more specific version of the equation that exposes the particular factors described in Section 3.4.1 and that uses the parameters defined in Table 3.1. In Section 3.4.4, we derive a formula for each term in Equation 3.2. We include an equation for bandwidth consumed in the absence of a cache, which expresses the upper bound on bandwidth consumption, in Appendix A.

3.4.4 Memory-Sensitive Consumption Equation

During ray tracing, the actual bandwidth consumed between two levels of memory is influenced by the data resident in the higher level, since data that is present does not need to be fetched from the lower level. The data that is present in the higher level memory is influenced by the coherence of the memory requests, which in turn is affected by the coherence of the rays traced. Tracing coherent rays reduces bandwidth consumed because these rays travel through the same nodes and are tested for intersection against the same geometry, which generate coherent requests to memory. Thus, we want our equation to contain terms that express ray coherence with respect to nodes and leaves, since that coherence directly determines the coherence of the memory requests.

We want terms that express both the ray coherence at internal nodes of the acceleration structure (c_n) and the ray coherence at leaves of the acceleration structure (c_l). We expect c_n to vary significantly for nodes at different levels of the acceleration structure, and providing a coherence term for each level of the structure would excessively complicate our equation. Since a leaf is guaranteed by construction to contain a volume no bigger than the internal nodes above it, we will use c_l, the coherence at the leaves, as a conservative estimate of the coherence at internal nodes. The value of c_l for a particular leaf l is exactly the total size of the MCRSs that: (1) end at l (l is the **tail leaf**, the last leaf touched by the MCRS), and (2) are processed atomically (i.e., no other MCRS is processed between two MCRSs for which l is the tail leaf). We use the expected value of c_l across all leaves touched by rays in the bandwidth equation. Figure 3.4.4 provides a visual reference for these

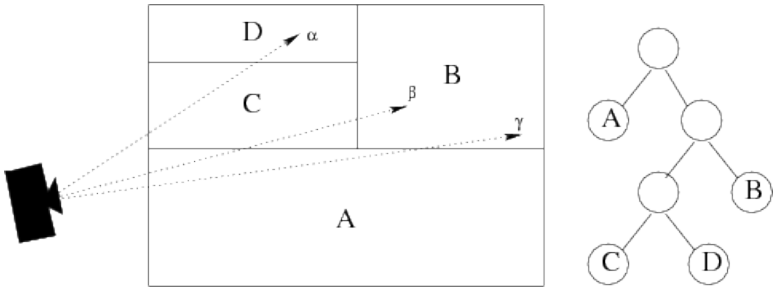

Figure 3.3: Memory-Coherent Ray Sets (MCRS) — This simple k-d tree demonstrates the MCRS concept defined in Section 3.3.1. The three rays shown are representative members of three MCRSs: ray α belongs to the MCRS $(root \rightarrow C \rightarrow D)$; ray β belongs to the MCRS $(root \rightarrow A \rightarrow C \rightarrow B)$; ray γ belongs to the MCRS $(root \rightarrow A \rightarrow B)$.

terms.

Armed with a term that defines ray coherence with respect to the acceleration structure, we are now prepared to derive the formulas for each term in the bandwidth equation, which we show in Sections 3.4.4.1–3.4.4.4.

3.4.4.1 Compulsory Bandwidth: C

Compulsory bandwidth C represents the first data load of each ray, node and object required for the ray tracing computation to complete. These are loads that *must* occur. The expected compulsory bandwidth consumed is the number of rays traced times the expected amount of data touched per ray:

$$C = R\left[COST_r + \mathrm{E}[\#\,CRS\ /\ ray]\,(n\,COST_n + g\,COST_g)\right] \tag{3.3}$$

3.4.4.2 Average Turnover Term: θ_a

To define the average turnover term θ_a, we must answer to two questions:

1. how much node and geometry data do we expect to revisit during the rendering pass?

2. is this data evicted from cache before each revisit, incurring a bandwidth cost to reload the data when it is next needed?

To answer the first question, we need the expected number of coherent ray sets that touch a leaf. We obtain this value from a profiling run by: (1) maintaining a ray mailbox at each leaf, (2) updating a global coherent ray set counter when the last ray traced matches the ray in the mailbox, and (3) dividing the global counter by the total number of leaves touched. Then, to obtain the expected number of times scene data is revisited, we multiply the expected number of MCRSs per leaf by the expected number of leaves touched during rendering.

To answer the second question, we must determine, between memory-coherent ray sets that end in a given leaf, how much *other* data (from outside the given coherent sets) will be loaded into cache. For this term, we use the expected leaf size. We explicitly account for leaves larger than the expected size with our large-leaf turnover and saturation-leaf turnover terms (θ_l and θ_s), since it is particularly these leaves that impact bandwidth consumption.

The data loaded between touches to a particular leaf, plus the given memory-coherent data sets, define the total data footprint of all the memory-coherent ray sets

for a given leaf. Required data that exceeds the cache size, divided by the cache size, expresses the expected rate at which the cache overflows per leaf. We will use the following terms to simplify notation in the formula:

- E[$MCRS_{bt}$] — the expected number of memory-coherent ray sets visited between two separate touches of a given leaf
- E[$rays_{bt}$] — the expected number of rays traced between touches of a given leaf
- B_{cache} — the size of the cache, the upper memory where data requests get stored, in bytes
- E[# *revisits / leaf*] — the expected number of times a leaf will be revisited, exactly one less than the expected number of MCRSs that visit a leaf. In algorithmic terms, the expected number of times that ray traversals through this leaf are interrupted by the traversal of rays that do not visit this leaf.

We have already expressed the expected bandwidth cost for a particular MCRS. To obtain the total data footprint, the complete working set, we need only to determine the expected number of MCRSs visited. The expected number of memory-coherent ray sets visited between two separate touches of a given leaf (E[$MCRS_{bt}$]) is represented by the expected number of rays processed between touches of the given leaf, divided by the expected size of a coherent ray set (see Figure 3.4.4.2 for an example frame buffer and MCRS).

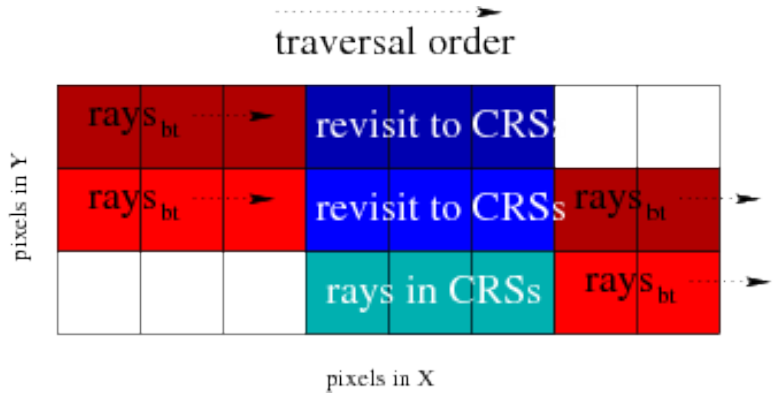

Figure 3.4: This portion of a frame buffer demonstrates how ray traversal affects memory use, both by revisits to an MCRS and by rays processed between visits to that MCRS. Assuming one primary ray per pixel, the pixels in blue represent nine rays that access the same series of MCRSs (we project the 3D leaves onto the 2D frame buffer for this example). The red pixels represent rays that do not touch the tail leaves of the MCRSs in question. Thus, for this example, $\mathrm{E}[|MCRS|] = 3$, $\mathrm{E}[rays_{bt}] = 5$, $(rows\ traced\ along\ Y) = 3$, and $\mathrm{E}[\#\ MCRS\ /\ ray]$ is the number of MCRSs projected onto the blue pixels.

$$\mathrm{E}[MCRS_{bt}] = \frac{\mathrm{E}[rays_{bt}] \times \mathrm{E}[\#\ MCRS\ /\ ray]}{\mathrm{E}[|MCRS|]} \tag{3.4}$$

We now need the expected number of rays processed between touches of a given leaf. If we assume a scan-line traversal and that the next touch of the given leaf begins at the same X value, then the expected number of rays processed between touches of a leaf is the number of rays traced along X (sampling rate along the X-axis), minus the expected size of a coherent ray set (note that this equation would need to be adjusted for other image-plane sampling patterns, such as tiling or a space-filling curve):

$$\mathrm{E}[rays_{bt}] = (rays\ traced\ along\ X) - \mathrm{E}[|MCRS|] \tag{3.5}$$

We can now write the formula for the average turnover term, Equation 3.6. In words, it is: the expected number of iterations across X (line 1); times the expected amount of data in the the current MCDS (line 2); times the expected size of all the memory-coherent data sets processed between two MCDS that include the current leaf (line 3); all minus the cache size (line 4). We assume that the total working set of the problem is larger than cache, so that $\theta_a \geq 0$.

$$\begin{aligned} \theta_a \quad = \quad & \frac{(rows\ traced\ along\ Y)}{\sqrt{R_a}} \\ & \times (\mathrm{E}[CRS_{bt}] + \mathrm{E}[\#\ CRS\ /\ ray]) \\ & \times (COST_r + \frac{n}{c_l} COST_n + \frac{g}{c_l} COST_g) \\ & - B_{cache} \end{aligned} \tag{3.6}$$

3.4.4.3 Large-Leaf Turnover: θ_l

We now consider the case when a ray pierces a leaf that contains more than the expected amount of geometry, but not so much that the geometry data cannot fit entirely in cache. We call these θ_l-leaves.

The expected cost of processing a θ_l-leaf is the expected **extra size** of the leaf (above the expected leaf size), times the probability that such a leaf will be touched during the traversal, times the expected number of leaves touched during rendering. We then multiply that amount by the expected MCRS size, because once

the leaf is hit, it is likely to be hit by subsequent rays in an MCRS. We do not count reuse here because a large leaf may be partially or completely evicted from cache between successive rays in an MCRS.

In our discussion of average turnover ta, we determined that the expression for the expected number of leaves touched is:

$$(\textit{rows traced along } Y) \times (E[CRS_{bt}] + \mathrm{E}[\#\, CRS \;/\; ray]) \tag{3.7}$$

Therefore, we need only to determine the expected extra size of a θ_l-leaf and the probability that it will be touched. The expected *extra* size of θ_l-leaves is the expected size of a θ_l-leaf minus the expected size of a leaf g. We want the extra size because we already accounted for the expected size in the ta term. The probability distribution for θ_l-leaves is the probability distribution for θ_a-leaves minus the probability distribution for a large leaf. We assume the expected large leaf size is at least g ($large\ leaf \geq g$), since the contribution of leaves up to the expected size is accounted for by the average turnover term θ_a. The tl term is defined by Equation 3.8.

$$\begin{aligned} \theta_l \;=\; & \frac{(\textit{rows traced along } Y)}{\sqrt{R_a}} \\ & \times \big(\mathrm{E}[CRS_{bt}] + \mathrm{E}[\#\, CRS \;/\; ray]\big) \\ & \times \big(\mathrm{P}(\theta_s\ leaf) - \mathrm{P}(large\ leaf)\big) \\ & \times \big(\mathrm{E}[size\ \theta_l\ leaf] - g\big)\, COST_g \\ & \times \mathrm{E}[|CRS|] \end{aligned} \tag{3.8}$$

3.4.4.4 Saturation-Leaf Turnover: θ_s

For small memory sizes, there can be leaves in the acceleration structure that contain more geometry than can be stored in upper memory at one time. Processing these leaves causes **saturation-leaf turnover**. When a ray pierces a saturation leaf, the data will evict all other data from cache. Processing the intersections will eliminate the benefit of coherent ray accesses because some or all of the geometry for the leaf will be reloaded for each ray[3]. Thus, no geometry is shared and bandwidth consumption is increased significantly beyond the consumption effects of large leaves. To model this effect, we identify each leaf that contains more geometry than can be stored in cache, then estimate how much bandwidth will be consumed both processing the ts leaves *and* reloading the data evicted by each ts leaf. We define ts in Equation 3.9.

$$\begin{aligned}
\theta_s \;=\; & \big(\textit{rows traced along } Y\big) \\
& \times \big(\mathrm{E}[CRS_{bt}] + \mathrm{E}[\#\ CRS\ /\ ray]\big) \\
& \times \big(1 - \mathrm{P}(\theta_s\ leaf)\big) \\
& \times \big(n\,COST_n + \mathrm{E}[size\ \theta_s\ leaf]\,COST_g\big) \\
& \times \mathrm{E}[|CRS|] \\
& + \big(\frac{R_a}{p} - 1\big)\,\big(\mathrm{E}[size\ \theta_s\ leaf]\big)\,COST_g
\end{aligned} \tag{3.9}$$

[3] processing rays in parallel, e.g. via SIMD instructions, maintains the coherent ray access benefit for the rays in a single parallel instruction block, but any coherence benefit is lost for successive blocks of rays.

3.5 Evaluating the Bandwidth Equation

To evaluate the accuracy of our bandwidth equation, we compare its consumption estimates against the measured bandwidth consumption of an experimental ray tracer with a simulated cache.

3.5.1 Experimental Ray Tracer Configuration

We have constructed an experimental ray tracer on which we can measure bandwidth consumed with respect to various ray traversal algorithms. The memory model implements a fully-associative cache with a least-recently-used (LRU) replacement policy. We use this configuration to minimize bandwidth consumption due to caching effects. In our implementation, we use the high-level cache model code from the SimpleScalar project [98].

We model eleven cache sizes in increasing binary powers from 4 KB to 4 MB. For these experiments, we focus only on memory behavior for data, ignoring memory effects for instructions. Each cache line is 64 bytes, which is the smallest size available in our simulation. One ray occupies an entire cache line, two acceleration nodes occupy one line (left and right siblings of a parent node), and one triangle occupies two cache lines. We choose these sizes both as reasonable estimates and so that there will be no address interference between objects in cache.

3.5.2 Test Scenes

We render four test scenes to provide a range of inputs for the parameters in Table 3.1. We use a k-d tree as the acceleration structure for each scene, built

according to the algorithm in Pharr and Humphrey's Physically-Based Ray Tracer (PBRT) [84]. Our algorithm is efficient, but it does not contain advanced optimizations, such as multi-level k-d tree traversal [91], that could further reduce memory accesses. Each scene is rendered at 1024×1024 resolution.

room — a room from a first-person perspective video game. This scene has a low total geometry count and a low used-to-total geometry ratio. The geometry is simple building geometry with some complexity from pipes and railing. The camera is located inside the acceleration structure.

soda hall — an architectural rendition of Soda Hall at Berkeley. This scene has a high total geometry count and a low fraction of visible geometry. The geometry is simple building geometry with some complexity from office furniture. The camera is located outside the acceleration structure.

sphereflake — a fractal-like sphereflake from Eric Haines's Standard Procedural Database (SPD) scene set. This scene has a high total geometry count and a high used-to-total geometry ratio. The geometry is complex and fractal-like, with many dense regions at the smallest spheres. The camera is located outside the acceleration structure.

grove — a collection of tree models (courtesy of Tim Purcell). This scene has a low total geometry count and a high used-to-total geometry ratio. The geometry is complex and organic, with many areas of geometric density. The camera is located outside the acceleration structure.

	room	**soda hall**	**sphereflake**	**grove**
R	1048576	1048576	1048576	1048576
$COST_r$	64	64	64	64
$COST_n$	32	32	32	32
$COST_g$	128	128	128	128
n	25.6	24.6	31.3	26.9
g	5.4	6.0	4.8	3.7
c_l (primary)	775.7	77.9	19.7	19.8
c_l (shadow)	775.7	28.6	4.6	10.9
E[# *MCRS* / *ray*]	9.8	1.4	12.2	21.7

Table 3.3: Values used for equation parameters in our tests.

3.5.3 Equation Evaluation Results

We now compare the equation's bandwidth consumption estimates against the measured bandwidth consumption of the experimental ray tracer. We present results for the four test scenes under both single-ray traversals and 8×8 packet tracing, a packet size that has been shown to achieve good memory system behavior [96]. The values for equation terms are presented in Table 3.3.

We present the comparison between the model estimates and our experimental measurements for single-ray traversal in Table 3.4 and for 8×8 packet traversal in Table 3.5. We report both the raw results and the percent error between the estimate and the measured value.

Our model provides a qualitative estimate for bandwidth performance. Both the model and the measured bandwidth exhibit threshold behavior as the cache size increases. The model predicts the thresholds accurately for **soda hall**, but it systematically predicts the thresholds at too-small cache sizes for **sphereflake** and

	primary rays											
cache	**room**			**soda hall**			**sphereflake**			**grove**		
size	est	act	Δ%	est	act	Δ%	est	act	Δ%	est	act	Δ%
4 KB	1835	9710	429	547	794	45	1736	5782	233	5488	12205	122
8 KB	2035	6375	213	372	524	41	1491	4245	185	4605	10401	126
16 KB	1090	851	28	360	332	8	1426	1727	21	4290	7790	82
32 KB	403	163	147	233	252	8	810	707	14	3287	3095	6
64 KB	220	160	38	170	205	20	414	602	45	1612	964	67
128 KB	67	139	107	120	143	19	302	596	97	595	939	58
256 KB	67	82	32	85	86	1	280	586	109	307	929	203
512 KB	67	68	1	80	71	13	273	555	103	247	903	266
1024 KB	67	68	1	80	71	13	273	188	45	247	821	232
2048 KB	67	68	1	80	71	13	273	178	53	247	342	38
4096 KB	67	68	1	80	71	13	273	178	55	247	218	13

	primary + shadow rays											
cache	**room**			**soda hall**			**sphereflake**			**grove**		
size	est	act	Δ%	est	act	Δ%	est	act	Δ%	est	act	Δ%
4 KB	1902	12293	546	627	916	46	3033	20803	586	5922	15982	170
8 KB	2102	4687	123	452	522	15	2788	17988	545	5038	11878	136
16 KB	1157	585	98	440	348	26	2723	10666	292	4724	7463	60
32 KB	470	226	108	313	273	15	2107	4293	104	3721	2806	33
64 KB	287	224	28	250	228	10	1692	2332	38	2046	989	107
128 KB	133	222	67	200	169	18	1516	2193	45	1026	963	6
256 KB	133	166	25	165	122	35	1366	2167	59	610	954	56
512 KB	133	132	1	160	94	70	1104	2136	93	494	928	88
1024 KB	133	132	1	160	94	70	1093	2052	88	494	850	72
2048 KB	133	132	1	160	94	70	1093	1640	50	494	427	16
4096 KB	133	132	1	160	94	70	1093	639	71	494	243	103

Table 3.4: Estimated bandwidth consumed (est) versus measured bandwidth consumed (act) for single-ray traversal. Bandwidth consumption is measured in megabytes. The model significantly underestimates bandwidth consumed by cache thrashing, such as for small cache sizes or for scenes where the acceleration structure is imbalanced (**sphereflake** , **grove**).

	primary rays											
cache		**room**			**soda hall**			**sphereflake**			**grove**	
size	est	act	Δ%	est	act	Δ%	est	act	Δ%	est	act	Δ%
4 KB	901	1754	95	471	511	8	938	3291	251	3518	7954	126
8 KB	473	468	1	159	271	70	456	1754	285	1147	4550	297
16 KB	195	189	3	121	147	21	418	742	76	772	1741	126
32 KB	109	86	27	99	91	9	340	337	1	627	633	1
64 KB	86	80	8	91	83	10	291	262	11	418	454	9
128 KB	67	80	19	85	81	5	277	235	18	290	319	10
256 KB	67	79	18	81	81	0	274	232	18	254	313	23
512 KB	67	78	16	80	81	1	273	232	18	247	313	27
1024 KB	67	68	1	80	71	13	273	230	19	247	309	25
2048 KB	67	68	1	80	71	13	273	205	33	247	297	20
4096 KB	67	68	1	80	71	13	273	178	53	247	268	8

	primary + shadow rays											
cache		**room**			**soda hall**			**sphereflake**			**grove**	
size	est	act	Δ%	est	act	Δ%	est	act	Δ%	est	act	Δ%
4 KB	945	11089	1073	551	1112	102	1817	18260	905	3788	14137	273
8 KB	540	6658	1133	239	591	147	1336	13664	922	1417	9901	599
16 KB	261	978	275	201	248	23	1298	6042	365	1042	6051	481
32 KB	175	151	16	179	128	40	1220	2840	133	897	2465	175
64 KB	153	144	6	171	106	61	1168	1415	21	688	536	28
128 KB	133	144	8	165	104	59	1146	980	17	560	344	63
256 KB	133	143	8	161	104	55	1127	860	31	508	338	50
512 KB	133	143	8	160	104	54	1094	842	30	494	338	46
1024 KB	133	141	6	160	97	65	1093	839	30	494	334	48
2048 KB	133	132	1	160	94	70	1093	821	33	494	325	52
4096 KB	133	132	1	160	94	70	1093	807	35	494	298	66

Table 3.5: Estimated bandwidth consumed (est) versus measured bandwidth consumed (act) for 8×8 packet traversal. Bandwidth consumption is measured in megabytes. The model still underestimates bandwidth consumed by thrashing at small cache sizes. However, the model better estimates bandwidth at larger cache sizes, since tracing coherent ray packets reduces the thrashing effects caused by imbalance in the acceleration structure.

grove. We believe this misprediction is due to the greater variance in leaf depth and leaf size in these scenes. The model is least accurate for **room**; we believe that the scene size lacks sufficient geometry to obtain good values for the statistical measures on which our model is based.

3.5.4 Discussion

The results presented in Table 3.4 and Table 3.5 show that even under our conservative ray coherence assumptions, the actual bandwidth consumed at small cache sizes exceeds our model's predictions. Our model tends to underestimate bandwidth consumption through the middle range of cache sizes tested, the range of sizes where most individual leaves fit in cache but a significant number of leaves (for example, all the leaves touched for a scan-line of rays) cannot yet be held at once. The equation underestimates the bandwidth for this range because the large-leaf turnover and saturation turnover components (θ_l and θ_s) are expected values taken over a wide interval, and thus contains high variance. The equation results exhibits steps at the sizes where the θ_s and θ_l contribution estimates reach zero (about 8KB and 128KB for **room** and **soda hall**, and about 16KB and 256KB for **sphereflake** and **grove**).

3.6 Summary

In this chapter, we have presented a model that qualitatively estimates the bandwidth consumption for ray tracing. Even under conservative assumptions for ray coherence, we find that our equation underestimates bandwidth consumed by

an actual tracer for small memory sizes and large scenes. This suggests that a bandwidth bottleneck exists: memory scarcity magnifies the impact of each memory request, and a scene with more data increases the likelihood of incoherent memory requests, which can ultimately saturate bandwidth.

Our model validation used an unoptimized research ray tracer with a range of reasonable hypothetical cache sizes. While this demonstrated that a bandwidth problem might exist for recursive ray tracing in general, we want evidence of a bandwidth bottleneck impacting a highly-optimized implementation running on actual hardware. It could be that the optimizations in a high-performance ray tracer cause it to use bandwidth more efficiently: for example, an optimized implementation might build a higher-quality k-d tree and traverse it more efficiently, which would result in fewer memory requests. In addition, scene data could be preprocessed in order to obtain more coherent memory accesses. In Chapter 4, we present a bandwidth consumption study using a high-performance ray tracer with actual hardware measurements that confirm a bandwidth bottleneck.

Chapter 4

Bandwidth Consumption Study for Single-Core Ray Tracing

Our model analysis in Chapter 3 suggests that a bandwidth bottleneck might exist for recursive ray tracers, but our validation was performed on an unoptimized ray tracer for hypothetical cache sizes. In this chapter, we measure the DRAM-to-cache bandwidth consumed by a highly-optimized, state-of-the-art recursive tracer for actual hardware configurations when tracing both coherent and increasingly incoherent (*divergent*) rays. We conclude that for current packet-tracing algorithms, bandwidth will not be a bottleneck for coherent rays, but that it will be a bottleneck for divergent rays. The trend in chip-multiprocessors for the next several years is for on-chip FLOPS to grow much faster than bandwidth to off-chip DRAM, so we expect that the bandwidth bottleneck will worsen in the future. This bottleneck is caused primarily by dramatically lower cache hit rates rather than by an increase in total working set, which suggests that substantial reductions in memory bandwidth consumption might be possible with a traversal algorithm that can create coherent groups from divergent secondary rays. In Chapters 5 and 6, we present our algorithms that demonstrate a substantial bandwidth savings over competing algorithms.

4.1 Overview

Performance of ray tracing systems might be limited by either raw FLOPS or by DRAM bandwidth. When designing algorithms, it is important to understand which potential bottlenecks are relevant. This chapter examines the potential DRAM bandwidth bottleneck by measuring the bandwidth consumed by packet-based recursive ray tracing, which is the *de facto* standard for contemporary systems. We report results for two cases that represent distinct classes of ray behavior: primary (camera) rays, which tend to be strongly coherent; and soft shadow rays, which can be slightly to significantly incoherent depending on the size, position and sampling of the light source. Soft shadow rays tend to be more coherent than rays generated by other secondary effects, such as ambient occlusion, diffuse reflection, or photon mapping, so we consider our soft shadow results to be a conservative estimate of expected performance.

Because the rays within a soft-shadow secondary-ray packet tend to diverge more rapidly than rays within a primary-ray packet, we expect that memory accesses for soft-shadow rays will less coherent, and in fact we find that this is the case. We conclude that today's popular algorithms are inefficient with respect to bandwidth usage for divergent secondary rays, and that these algorithms should be adapted or replaced to improve their utilization of the memory hierarchy.

The remainder of the chapter is organized as follows: we describe related work in Section 4.2. In Section 4.3 we describe our experimental method. We present our results in Section 4.4, and we summarize the chapter in Section 4.5.

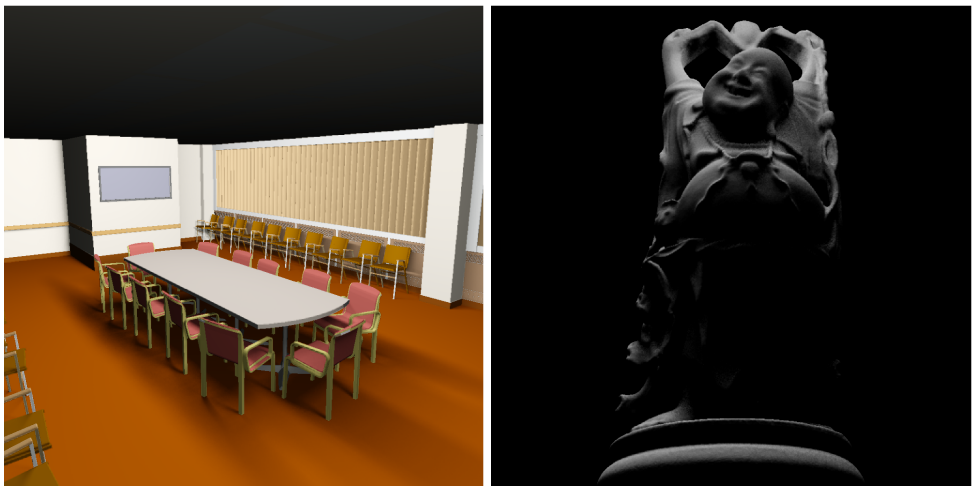

Figure 4.1: The scenes used in this chapter: **conference** (283K total triangles, 54K visible); and **statue** (1088K total triangles, 369K visible)

4.2 Related Work

There are several published results for the bandwidth requirements of tracing primary rays and hard shadows. Schmittler et al. [96, 97] and Woop et al. [121] report primary ray and hard shadow memory bandwidth consumption on dedicated ray tracing hardware. Benthin et al. [13] measure memory traffic for primary rays on the IBM CELL processor. Purcell et al. [88] measure bandwidth between on-board RAM and processor for GPU-based ray casting. When scaled by cache size, our measured bandwidth consumption for primary rays and hard shadows are similar to these related results.

We are aware of only a few published bandwidth results for ray tracing with divergent secondary rays. Purcell et al. [88] report bandwidth consumption for specular reflection and for path tracing on a GPU. They find that tracing secondary

rays increases DRAM to processor memory traffic 3× to 4× over primary ray traffic. Pharr et al. [85] describe bandwidth consumption between main memory and disk for a complex scene rendered with a Monte Carlo global illumination simulation. Though this paper deals with a lower level of the memory hierarchy, they find significant excess bandwidth consumption due to thrashing data from disk to main memory. We will compare their findings with ours in Section 4.5.

4.3 Experimental Method

In this section, we describe our experimental set-up. We model the cache configurations of three current hardware architectures in order to test the bandwidth consumed by coherent primary rays and divergent soft shadow rays. For each configuration, we render a low-complexity scene and a high-complexity scene. We provide additional details below.

4.3.1 Cache Configurations

We use cache configurations that correspond to three current processors: two traditional CPUs and a chip-multiprocessor (CMP) system. For our CPU selections, we use a current processor from Intel and AMD: an Intel Prescott core with 975x chipset [51] and an AMD Toledo core and memory controller [1]. We use the Sun Niagara processor [61] for our CMP model (though we assume floating-point support), since it has hardware multi-threading, a large L2, and high DRAM to L2 bandwidth. Note that Niagara has a 3MB L2 cache, but our cache simulator forces us to use power-of-two sizes. We conservatively use a 4MB L2. The system

Processor	L1 cache configuration	L2 cache configuration	DRAM to L2 bandwidth
Intel Prescott	16KB, 8-way, 64B lines	1MB, 8-way, 64B lines	10.7 GB/s (975x chipset)
AMD Toledo	64KB, 2-way, 64B lines	1MB, 16-way 64B lines	8.0 GB/s
Sun Niagara (constructive)	8KB, 4-way, 16B lines	4MB, 12-way, 16B lines	20 GB/s
Sun Niagara (destructive)	2KB, 4-way, 16B lines	128KB, 12-way, 16B lines	20 GB/s
IBM CELL			25.6 GB/s
GeForce 7900 GTX			51.2 GB/s
SaarCOR			1 – 2 GB/s

Table 4.1: System Configurations — the first four rows contain the systems we simulate: two traditional CPUs and two cases for a CMP system. We include other rendering hardware for bandwidth comparison only.

configurations are summarized in Table 4.1.

We model two configurations for the Niagara: a constructive cache interference pattern, where all processing resources can share cached data; and a destructive cache interference pattern, where no processing resource shares data. For the destructive interference pattern, we partition memory-system resources evenly to each thread on each core. These two interference patterns establish upper and lower limits on memory system performance.

4.3.2 Scenes

We use two scenes in our survey (shown in Figure 4.1), each rendered at 1024×1024 resolution. We use **conference**, since it was used in previously re-

ported performance results for the tested ray tracer [91]. This permits us to compare our results against previous work. We also use a scene with the Stanford Buddha statue so that we have more scenes with many visible triangles. We trace each scene using two different methods: primary rays only, and primary rays with soft shadows. We choose these methods as representative samples from coherent and divergent classes of ray complexity to help judge the capacity of each memory system. We generate one frame's worth of memory traffic data for each scene. We use the single-frame data to estimate animation memory traffic at 60 fps.

4.3.3 Ray Tracing Algorithms

Our work uses a state of the art ray tracer as the basis for our architecture study. At the time of its publication, the MLRTA system [91] achieved the best published frame rates on CPUs for static scenes.

In this system, primary rays are traced in packets of 16 rays, generated through contiguous 4×4 pixel blocks. Soft shadow sampling is performed in 4×4 packets that also correspond directly to hit points for a primary ray packet. Shadow ray origins are generated randomly on the surface of the area light, with a separate origin created for each ray in the packet. Nine soft shadow samples per primary-ray hit point are generated per sampling round. If the surface is too rough, only one sampling round is performed. Otherwise, soft shadow sampling continues until sample variance is under a given threshold. Surface roughness is determined by taking the cosine of the angle between surface normals at each pair of hit points from the primary ray packet and comparing the minimal cosine value against a threshold.

4.3.4 Measurement Methodology

We create a trace of memory requests made by the MLRTA ray tracer when rendering each combination of scene and ray tracing algorithm. We record a cache read for ray and node data at each traversal step and a read for ray and geometry data at each intersection step. We never record cache writes. We use the Dinero IV cache simulator [31] to model each memory system. This light-weight simulator provides cache usage statistics without modeling functionality or providing timing estimates.

Our measurements conservatively estimate the memory system resources required to process the sample loads because we only model data reads during ray traversal and intersection. We do not model acceleration structure generation, instruction cache traffic, shading or writing the image. As such, we expect our measurements to serve as a lower bound for the memory-system requirements for ray tracing systems.

We report our findings both in terms of bandwidth consumed and in terms of cache efficiency. We define cache efficiency as compulsory bandwidth consumed (i.e. total size of working set) divided by total bandwidth consumed. With this measure, we can estimate how much cache utilization could be improved.

4.4 Results

We traced coherent camera rays and divergent soft-shadow rays on **conference**, a scene with relatively few visible triangles (54K visible) and on **statue**, a

machine	scene	primary rays only L2 to L1 traffic	DRAM to L2 traffic	compulsory traffic
Intel Prescott	conference	14 649 856	4 836 416	4 833 024
	statue	63 664 000	32 026 688	32 001 472
AMD Toledo	conference	11 039 616	4 835 776	4 833 024
	statue	59 739 840	32 011 072	32 001 472
Sun Niagara (constructive)	conference	13 726 672	4 214 736	4 214 736
	statue	49 424 016	28 732 560	28 732 560
Sun Niagara (destructive)	conference	40 396 432	5 420 320	4 214 736
	statue	78 355 200	41 465 696	28 732 560

machine	scene	primary + soft shadows L2 to L1 traffic	DRAM to L2 traffic	compulsory traffic
Intel Prescott	conference	902 914 496	28 859 264	8 705 280
	statue	16 886 719 744	9 090 462 080	54 827 456
AMD Toledo	conference	371 575 744	29 347 904	8 705 280
	statue	16 121 283 648	8 795 612 672	54 827 456
Sun Niagara (constructive)	conference	977 741 632	8 230 672	8 220 272
	statue	10 673 545 664	1 929 019 776	51 919 504
Sun Niagara (destructive)	conference	2 785 107 632	106 545 280	8 220 272
	statue	12 070 645 792	9 737 228 000	51 919 504

machine	scene	consumption increase for divergent rays L2 to L1 traffic	DRAM to L2 traffic	compulsory traffic
Intel Prescott	conference	61.63x	5.97x	1.80x
	statue	265.25x	283.84x	1.71x
AMD Toledo	conference	33.66x	6.07x	1.80x
	statue	269.86x	274.77x	1.71x
Sun Niagara (constructive)	conference	71.23x	1.95x	1.95x
	statue	215.96x	67.14x	1.81x
Sun Niagara (destructive)	conference	68.94x	19.66x	1.95x
	statue	154.05x	234.83x	1.81x

Table 4.2: Memory Traffic — total data traffic in bytes for a single frame of (**conference** and **statue**). Dividing compulsory traffic by total traffic provides our efficiency measurement for each cache level. Compulsory traffic is slightly lower for the Niagara tests because it has shorter cache lines.

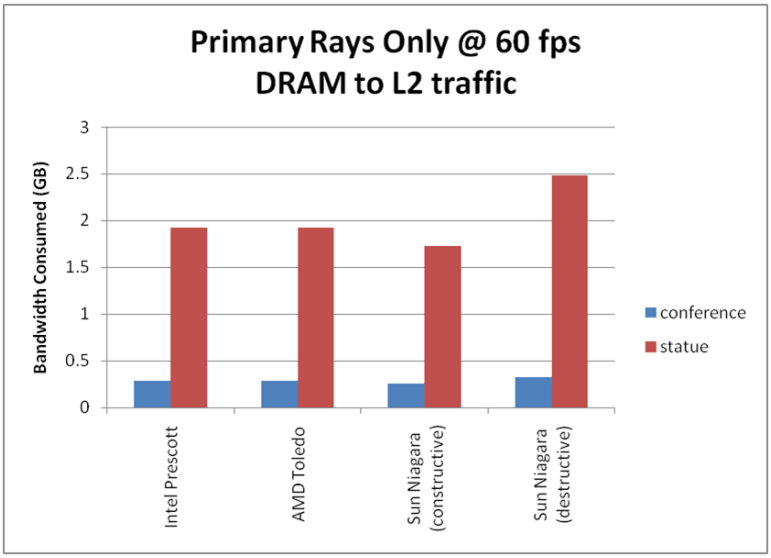

Figure 4.2: DRAM to L2 data traffic in bytes for primary rays, extrapolated to 60 fps. The traffic is well within current DRAM to L2 bandwidth rates in Table 4.1 (8 GB – 20 GB).

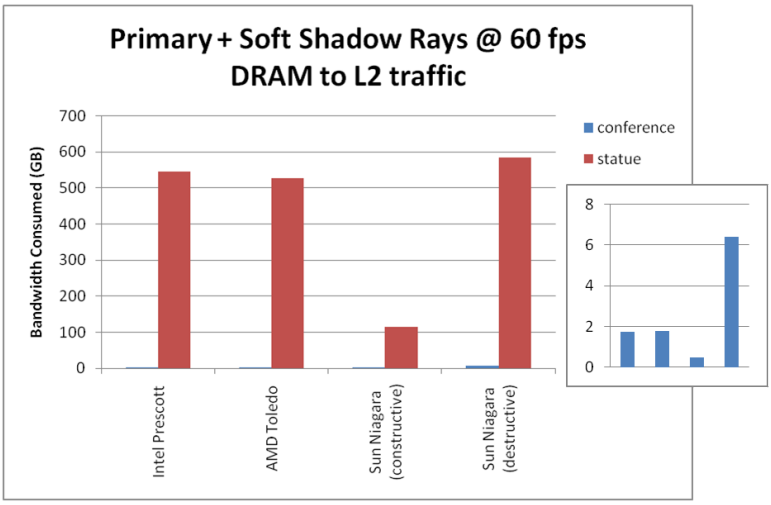

Figure 4.3: DRAM to L2 data traffic in bytes for primary + soft shadows, extrapolated to 60 fps. When there are few visible triangles (**conference**), traffic is within current DRAM to L2 bandwidth rates (8 GB – 20 GB). When there are many visible triangles (**statue**), traffic exceeds current bandwidth rates by 10× or more.

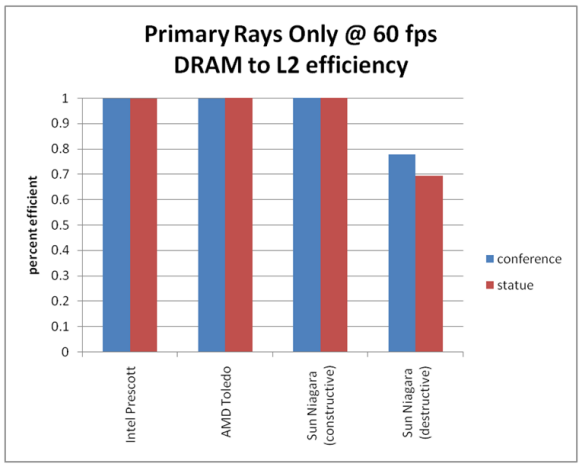

Figure 4.4: DRAM to L2 efficiency for primary rays only — when tracing only primary rays, bandwidth between DRAM and L2 is used efficiently, even for the 128KB L2 of the Sun Niagara (destructive) case.

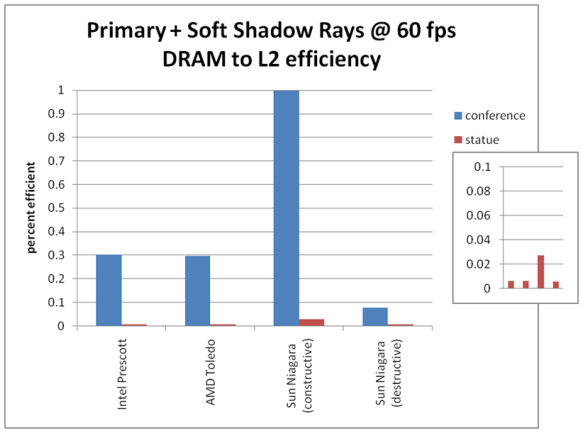

Figure 4.5: DRAM to L2 efficiency for primary + soft shadows — when tracing soft shadows, bandwidth efficiency is maintained when the working set can be maintained in cache (Sun Niagara (constructive)). When the working set cannot be maintained in cache, cache efficiency degrades significantly (**conference**) or catastrophically (**statue**).

scene with significantly more visible triangles (389K visible). As the data in Table 4.2 shows, rendering with divergent secondary rays can increase bandwidth consumed between main memory and L2 by several factors; for scenes with many visible triangles, rendering with divergent secondary rays can increase bandwidth consumed by one to two orders of magnitude.

In Figure 4.2 and Figure 4.3, we extrapolate these data to simulate rendering at 60 fps. Bandwidth demand for scenes with few visible triangles is within the current memory-to-core bandwidth rates for general-purpose hardware reported in Table 4.1, but bandwidth demand for scenes with many visible triangles exceeds current bandwidth rates by more than an order of magnitude.

The dramatic increase in bandwidth consumption for divergent secondary rays cannot be explained solely by the increase in the size of the working set, which is measured by compulsory memory loads. By comparing the total bytes loaded to the compulsory byte loads (Figure 4.4 and Figure 4.5), we see that there is a large difference in how efficiently the L2 cache is used. Primary rays produce high L2 hit rates, whereas divergent secondary rays produce much lower hit rates. The Niagara results are particularly evocative. First, consider the results for few visible triangles. In the constructive interference case, where the entire 4MB of cache is available for scene data, the L2 hit rate is high; in the destructive interference case, where only a fraction of the L2 is available, the L2 hit rate is considerably lower. When we consider the results for many visible triangles, we see that the hit rate is miserable for both L2 cases.

4.5 Summary

We have examined the DRAM bandwidth bottleneck for ray tracing systems, and we conclude that recursive traversal algorithms are inefficient with respect to divergent secondary rays. This inefficiency is a result of poor cache management rather than a dramatic increase in the working set. Future ray tracing systems must either maintain better ray coherence or increase available cache to maintain the entire working set. Since new processor designs have less cache per core, ray tracing algorithms for such systems should be modified or replaced to achieve better utilization of the memory hierarchy.

A queueing ray tracing algorithm might provide sufficient secondary ray coherency. Pharr et al. [85] compare the memory performance of a recursive tracer and a queueing tracer rendering a 9.6M triangle scene with Monte Carlo path tracing. On a system with 325MB RAM dedicated to geometry data, the recursive tracer generates 120MB of compulsory geometry traffic and 2.1GB of additional traffic from incoherent geometry accesses, which is only 5% efficient by our measure. Their ray queueing strategy eliminated excess geometry traffic while incurring 70MB of new ray traffic, which is 63% efficient (including ray traffic) and over 12$\times$ more efficient than the recursive tracer. With only 50MB of RAM dedicated to geometry data, the queueing tracer generated 938MB excess geometry traffic and 70 MB of ray traffic, which is 12% efficient and still more than twice as efficient as the recursive tracer (which had 6.5$\times$ more memory). For such a technique to be viable, it should be compatible with modern surface shaders and SIMD-based instruction optimizations. The additional memory traffic generated by maintain-

ing ray state must be factored against any savings from lower geometry traffic. In Chapters 5 and 6, we demonstrate that by using a queueing tracer, we can create dynamic scheduling algorithms that efficiently use the memory hierarchy.

Chapter 5

Dynamic Scheduling for a Single Core

As we discussed at the end of Chapter 4, queueing rays can be a potent means to increase memory system efficiency during ray tracing. We want to determine what algorithmic adjustments to Pharr et al.'s algorithm [85], if any, are required to target higher levels of the memory hierarchy. In this chapter, we examine dynamic ray scheduling from the perspective of a single processing core of a multi-core chip. We will broaden our scope to distributed memory multi-processor systems in Chapter 6.

5.1 Overview

In the last chapter, we showed the stress that incoherent rays, and the incoherent memory requests they generate, place on processor cache and on system RAM. Pharr et al. built an ray tracer [85] that improves memory efficiency by processing rays according to their location in scene space, independent of their place in a traversal of the ray tree. In this algorithm, rays traverse a uniform grid, and they are enqueued at any grid cell that contains geometry. When a cell is selected for processing, all rays enqueued at that cell are tested for intersection against geometry in the cell. Rays that do not intersect an object are traversed 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 the 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 workloads 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 chapter, we present an algorithm that schedules the processing of rays by *actively managing* both ray and geometry state to maximize cache utilization and bandwidth efficiency. As we described in Chapter 2, Pharr et al.'s queueing algorithm and Whitted's recursive algorithm are two points in a spectrum of ray scheduling algorithms that control which rays are traced together and against which data those rays are traced. In this view, our new approach generalizes both algorithms, selecting the appropriate point in the spectrum based on the available resources of the host architecture. To demonstrate its feasibility, we use a detailed simulation to show that our algorithm significantly reduces the amount of geometry loaded when traversing incoherent secondary rays, with only moderate overhead to handle ray state. Our algorithm performs best when the amount of visible geometry is much larger than available cache and when that geometry is traced with incoherent rays. For example, on the 164K triangle **grove** scene rendered with hard shadows and diffuse reflections with 512KB cache available, our algorithm reduces DRAM-to-L2 bandwidth consumption 7.8$\times$ compared to packet ray tracing. We

conclude that our notion of *dynamically scheduled rays* provides data access patterns that are spatially coherent both in the scene and in the machine address space. Our algorithm can be combined with current ray tracing optimizations for coherent ray data, and, unlike those optimizations, it promises to scale for use with complex lighting models.

The rest of the chapter proceeds as follows. First, we describe specific related work in Section 5.2. Then, in Section 5.3, we describe how the insights from our bandwidth model led to an initial dynamic scheduling algorithm for primary rays and present initial results that demonstrate its ability to outperform both recursive and Pharr-like traversals. We then present the full the algorithm in Section 5.4. We explain our testing methodology for the full algorithm in Section 5.5, and we present results in Section 5.6.

5.2 Related Work

Our work builds on several previous ray scheduling schemes to create an algorithm in which both geometry and rays are actively managed. We now discuss this and other related work to better distinguish the contributions of our algorithm.

As we described in Section 2.1.3, all recursive ray tracers implement some form of Whitted's original recursive ray tracing algorithm [119]. Techniques like packet tracing [115] can increase the number of rays active in the system at once, but 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 efficiently

use the memory system. Reordering tracers [18] can reform packets between ray generations, but the active ray state remains fixed during traversal of each packet.

Pharr et al. [85], take an approach quite different from the Whitted tracing model. Their algorithm permits ray queueing by using a linear formulation of the rendering equation [56], where the outgoing radiance is the weighted sum of the incoming radiance. This formulation computes radiance as rays are traced rather than accumulating it on the recursion stack. As a consequence, many more rays can be active in the system at once, providing greater opportunity to trace rays coherently. As we discussed in Section 4.5, Pharr et al.'s algorithm improves secondary ray coherency and reduces traffic between disk and RAM when rendering large models on a single machine.

Since Pharr et al.'s algorithm targets disk-to-RAM efficiency, there are aspects of it that are poorly suited to manage RAM-to-cache traffic. The acceleration structure is a uniform grid, which, because it is non-adaptive, makes it difficult to constrain the amount of geometry at any particular cell. This variance can be masked at the RAM level, but at the cache level it complicates effective scheduling. Also, while rays can be enqueued under Pharr et al.'s algorithm, there is no bound on the number of active rays. Instead, the algorithm quickly descends into the ray tree, so many secondary rays of many ray generations can be active in the system at once. Again, while this technique may be effective when considering disk-to-RAM traffic, the ray state explosion that results can cause serious 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 other prior implementations of queueing algorithms. Dachille and Kaufman [25] implemented a queueing traversal for dedicated volume rendering hardware. In this system, rays are collected at each cell of the volume, much the way that rays are collected in Pharr et al.'s uniform grid. Cells are then scheduled for processing as in Pharr et al.'s algorithm. They were able to achieve interactive frame rates using a hardware simulator Because they use direct volume rendering, there is no significant geometry traffic per cell: the system loads the eight vertices of the cell and tri-linearly interpolates each sample along each ray, so their system has virtually no geometry traffic to manage. Steinhurst et al. [99] use Pharr-like queueing to obtain better cache performance for photon mapping. Like Pharr et al., 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 might perform better on this task because it actively manages ray state, and our results on diffuse reflection rays support our hypothesis.

5.3 Dynamic Scheduling for Primary Rays Only

In Chapter 3, we discovered that incoherent ray traversals of large leaves, terminal nodes of the acceleration structure that contain more geometry than average, can dramatically increase incoherent memory requests. These requests can ultimately saturate bandwidth between main memory and processor cache. Using the equations presented in Section 3.4, we develop an algorithm that mitigates the effects of large leaves on memory system efficiency. Specifically, we design our algorithm to *enqueue* rays at these large leaves to increase the number of rays that use

the leaf data per load. To use our equation terms, we create new memory-coherent ray sets (MCRSs) at each large leaf. We want to maximize the size of each MCRS. This new algorithm exploits the spatial coherence of all rays, similar to the scheduling grid queues described in Pharr et al. [85], rather than just the spatial coherence of rays in a particular packet.

We find that this new algorithm consumes less bandwidth than packet tracing when cache resources are scarce. In addition, when there is little or no cache pressure, the new algorithm avoids the overhead of Pharr et al.'s algorithm, since it does not enqueue at each leaf. We call this new algorithm *dynamic ray scheduling* since the queue points are determined dynamically according to the acceleration structure behavior and the cache resources available on the particular hardware.

5.3.1 Creating the Dynamic Ray Scheduling Algorithm

The inspiration for the dynamic ray scheduling algorithm came from an analysis of the large leaf turnover equation, Equation 3.8. If the algorithm reduces the number of times the data at a large leaf is requested, it will reduce the memory effects of that large leaf on the rest of the rendering. We walk through the steps of the algorithm below.

- Trace each ray packet recursively [115]. If the packet does not hit any large leaves, its traversal is identical to a recursive packet-based traversal.
- If a large leaf is encountered, enqueue the current packet by adding it to a queue of packets maintained at that leaf. Begin tracing the next packet of

primary rays.

- After all primary rays have been traced:
  - If no packets have been enqueued, rendering is complete.
  - If packets have been enqueued, select a leaf that contains queued packets. For this work, we choose the leaf closest to the camera so that a leaf with enqueued packets will be touched only once. The full algorithm uses a different queue selection method.
  - Perform ray-object intersection tests for the objects stored at the leaf. We perform the intersection tests in ray-order (test one ray against all geometry before considering the next ray), but other intersection orderings are possible (object-order, for example).
  - For each packet that still has active (not yet intersected) rays, continue a packet traversal for the packet as though it had not been enqueued at the current leaf. If a packet encounters another large leaf, queue the packet at the new leaf and begin tracing the next packet from the current leaf.
  - Continue selecting leaves with queued packets until all rays have been traced.

We now walk through a toy example of the algorithm in action. Imagine an acceleration structure that contains one large leaf. To simplify the presentation, say that 10% of camera rays hit this leaf and are enqueued. The remaining 90% of camera rays are traversed exactly as in a packet traversal. Now, we check for any

enqueued rays. There is one leaf with a non-empty queue, so it is selected. We now perform intersection tests for the rays and geometry at the selected leaf. Let us say that 40% of the queued rays are successfully intersected. For each of the remaining rays (6% of all the original camera rays), we resume a packet traversal, starting at the ray's exit point from the large leaf. Because there are no more large leaves, each ray will complete its traversal without being enqueued again.

This traversal minimizes visits to each large leaf. Further, the traversal increases ray coherence because it traverses one set of queued rays to completion[1] before processing another queue. In the next section, we will describe how to select the size threshold for large leaves, and we will present our bandwidth consumption results for the dynamic traversal.

5.3.2 Primary Ray Scheduling Results

In this section, we compare the bandwidth consumption of primary rays for a dynamic scheduling traversal against the consumption for single-ray traversal, a 2×2 packet traversal, an 8×8 packet traversal, and an "always enqueue" traversal similar to Pharr et al., but that uses the same queue selection metric as our dynamic traversal (proximity to the camera). This Pharr-like algorithm always enqueues rays at a particular node depth to approximate their the scheduling grid, whereas our algorithm enqueues only at large leaves. We find that when cache resources are scarce, our dynamic scheduling traversal reduces bandwidth consumption com-

[1] By completion, we mean the ray was successfully intersected, enqueued at another large leaf, or exited the acceleration structure.

pared to each recursive traversal, and it remains competitive with the "always enqueue" traversal. When cache resources are plentiful, our algorithm is competitive with the recursive traversals, and it does not suffer from the same overhead costs as the "always enqueue" traversal.

For these experiments, we modified our experimental ray tracer described in Section 3.5.1 to perform a dynamic traversal. We then rendered each of the four scenes described in Section 3.5.2. We will present results from camera locations and directions that are representative of the performance of each traversal on each scene.

5.3.2.1 Selecting Where to Enqueue Rays

Since we define a large leaf heuristically in Section 3.4.4.3, we test a range of leaf sizes to measure the impact of selecting the leaf size at which rays will be enqueued. Our tests range from 4 KB, where rays are enqueued at every leaf, to 64 KB, where rays are never enqueued. We find that the ideal point to start deferring rays occurs when: (1) there are leaves large enough to cause significant turnover in cache, and (2) enough rays hit those leaves to counter the queueing overhead. For our tests, we found the best queue point to be at leaves that contain geometry equivalent to at least one-quarter of the total data cache. We used this queue point for our results.

single rays

cache size	room rec	room enq	room Δ%	soda hall rec	soda hall enq	soda hall Δ%	sphereflake rec	sphereflake enq	sphereflake Δ%	grove rec	grove enq	grove Δ%
4 KB	35.2	24.9	41	5.9	4.1	44	56.6	28.1	101	64.8	40.2	61
8 KB	7.9	4.4	80	4.3	2.8	54	45.1	29.7	52	46.9	20.3	131
16 KB	2.1	1.7	24	3.5	1.7	106	19.4	12.8	52	22.7	7.8	191
32 KB	1.6	1.6	0	2.9	1.8	61	8.8	8.6	2	6.4	5.4	18
64 KB	1.6	1.6	0	2.2	2.2	0	8.0	8.0	0	5.2	5.2	0

2×2 packets

cache size	room rec	room enq	room Δ%	soda hall rec	soda hall enq	soda hall Δ%	sphereflake rec	sphereflake enq	sphereflake Δ%	grove rec	grove enq	grove Δ%
4 KB	13.9	24.2	-74	4.2	3.8	10	30.2	25.7	18	45.0	39.7	13
8 KB	4.5	3.8	18	3.1	2.6	19	22.5	19.3	16	26.2	18.8	39
16 KB	1.5	1.4	7	1.9	1.4	36	10.4	8.7	19	11.1	6.2	79
32 KB	1.3	1.3	0	1.7	1.4	21	5.8	5.8	0	4.0	3.7	8
64 KB	1.3	1.3	0	1.5	1.5	0	5.0	5.0	0	3.3	3.3	0

8×8 packets

cache size	room rec	room enq	room Δ%	soda hall rec	soda hall enq	soda hall Δ%	sphereflake rec	sphereflake enq	sphereflake Δ%	grove rec	grove enq	grove Δ%
4 KB	17.8	23.8	-33	5.5	4.5	22	47.6	30.6	56	59.0	44.7	32
8 KB	3.9	4.1	-5	3.2	2.6	23	24.4	18.5	32	24.2	18.0	34
16 KB	1.2	1.2	0	1.8	1.2	50	7.7	6.8	13	6.6	5.3	24
32 KB	1.1	1.1	0	1.2	1.2	0	4.0	4.3	-8	2.4	2.5	-4
64 KB	1.1	1.1	0	1.2	1.2	0	3.2	3.2	0	1.8	1.8	0

Table 5.1: Bandwidth consumed by a recursive traversal (rec) versus our queueing variant (enq). Bandwidth consumption is measured in megabytes.

5.3.2.2 Measured Bandwidth Consumption

We compare the bandwidth consumption of a dynamic traversal against the bandwidth consumption of several other traversal algorithms. For single-ray and packet traversals, we make pair-wise comparisons of the standard traversal versus its dynamic counterpart in Table 5.1. We also compare the bandwidth consumption of our algorithm against an "always enqueue" algorithm in Table 5.2. We present the bandwidth consumption measurements graphically in Figures 5.3 – 5.4.

	single rays											
cache	**room**			**soda hall**			**sphereflake**			**grove**		
size	always	large	Δ%	always	large	Δ%	always	large	Δ%	always	large	Δ%
4 KB	10.7	24.9	-133	4.1	4.1	0	22.4	28.1	-25	38.2	40.2	-5
8 KB	8.4	4.4	91	3.2	2.8	14	17.5	29.7	-70	18.9	20.3	-7
16 KB	6.2	1.7	265	2.0	1.7	18	10.1	12.8	-27	8.3	7.8	6
32 KB	5.7	1.6	256	2.0	1.8	11	9.3	8.6	8	6.4	5.4	18
64 KB	5.7	1.6	256	1.9	2.2	-16	8.8	8.0	10	6.1	5.2	17

Table 5.2: Bandwidth consumed by always queueing (always) versus queueing at only large leaves (large). Bandwidth consumption is measured in megabytes.

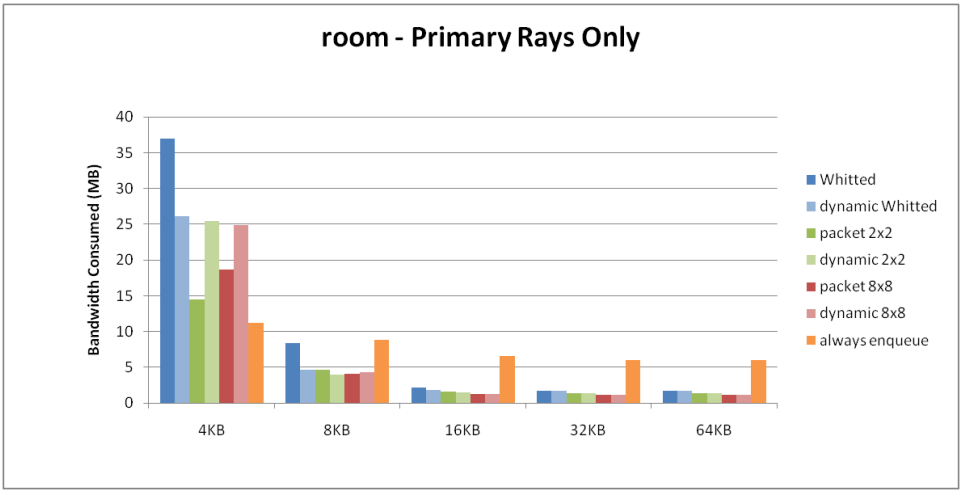

Figure 5.1: Bandwidth consumed by primary rays for **room**.

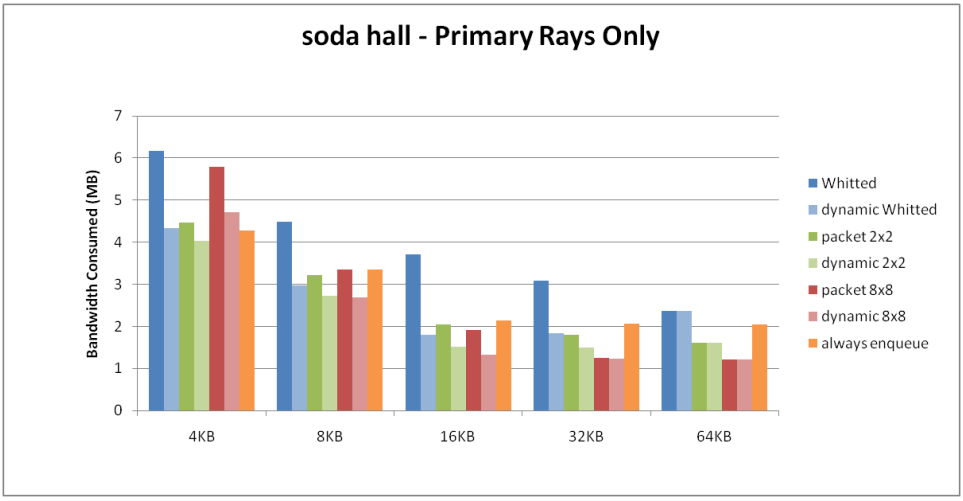

Figure 5.2: Bandwidth consumed by primary rays for **soda hall**.

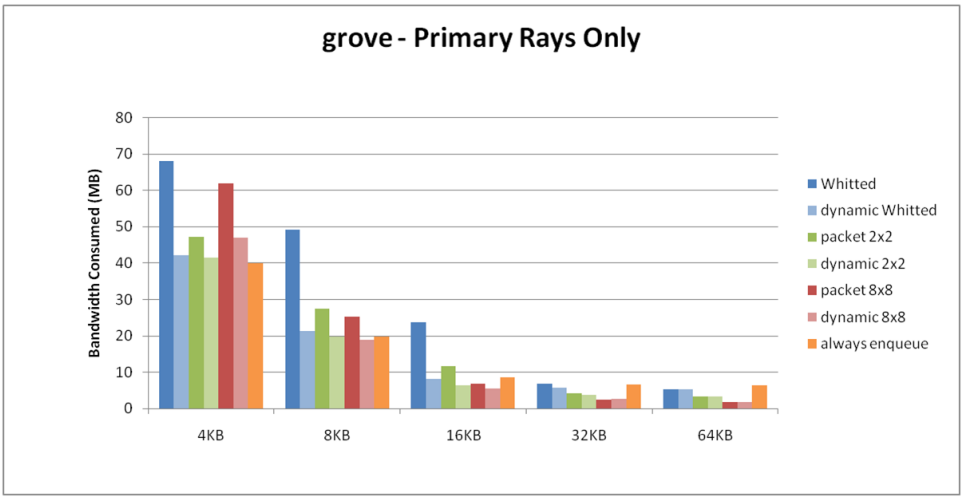

Figure 5.3: Bandwidth consumed by primary rays for **grove**.

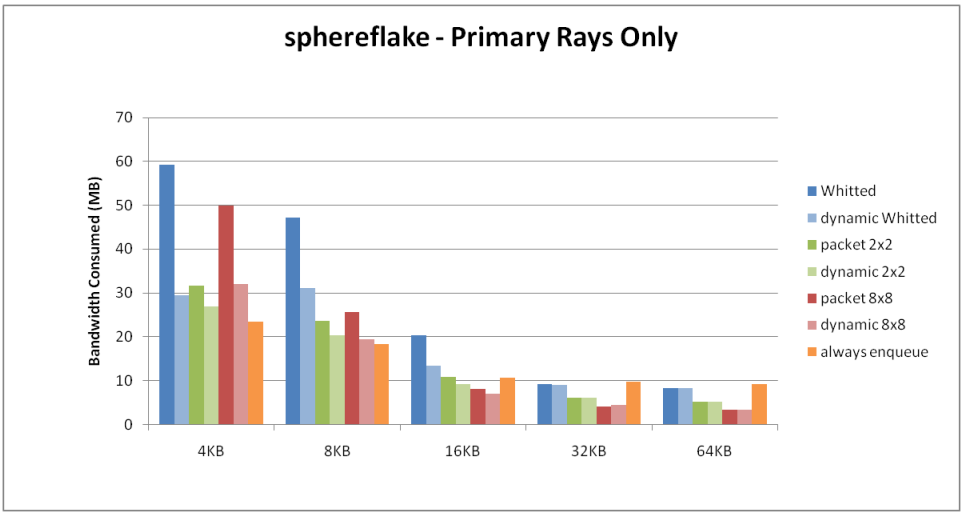

Figure 5.4: Bandwidth consumed by primary rays for **sphereflake**.

5.3.3 Discussion

Our results show that using a dynamic scheduling traversal can reduce bandwidth consumed for all scenes tested. The largest bandwidth savings came from making the single-ray traversal dynamic. This result was expected, since a single-ray traversal benefits only from image coherence and not from any ray coherence. Our algorithm also improves the bandwidth consumed by packet traversals when cache resources are scarce. While a packet traversal benefits from ray coherence, it only exploits memory coherence between packets when the cache is large enough to hold the entire unit of work for the packet. Our algorithm captures this lost coherence by increasing the work done for each large leaf.

As cache size increases, the benefit of our algorithm decreases for all scenes. This is by design, since as cache size increases fewer leaves are deemed large, and thus fewer rays are enqueued. When no large leaves are found, our algorithm per-

forms as a recursive algorithm, with slight overhead cost to check for large leaves. It does not suffer the more significant overhead measured for the "always enqueue" strategy.

Our algorithm performed worst on the **room** scene, though bandwidth savings were realized for the smallest cache sizes. We attribute this performance to the low geometry load of the scene in general[2] and thus to there being few large leaves present. The bandwidth savings at these few leaves is not sufficient to amortize the queueing overhead.

5.4 Complete Dynamic Scheduling

Our complete dynamic scheduling algorithm actively manages ray and geometry state to provide better cache utilization and lower bandwidth requirements, which enables faster execution. Our algorithm is rooted in two concepts: rays can be traced independently (non-recursively), and rays can be enqueued at regions in scene space where the geometry in that region fits completely in available memory. Taken together, these concepts permit tight control of 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

[2] **room** is our smallest test scene and it is overly simplified in two respects: first, it was created for rendering on a z-buffer rasterizing architecture, which encourages low geometric complexity (since all geometry must be touched during the algorithm's execution); and second, there are no characters in the scene that would add significant geometric complexity.

lowest level of processor cache itself. We conservatively treat this as a dedicated cache; contention among cores for a shared cache will further decrease bandwidth utilization (see the Sun Niagara results in Section 4.4) and increase the value of our approach. 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 hardware that we target.

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 acceleration structure to adapt to varying densities of scene geometry directly affects schedule quality by limiting the flexibility that 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 different acceleration structures is beyond the scope of this chapter.

5.4.1 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, which adjusts the primary-only algorithm described in Section 5.3. We then expand the discussion to include secondary rays. Section 5.4.3 contains an implementation sketch.

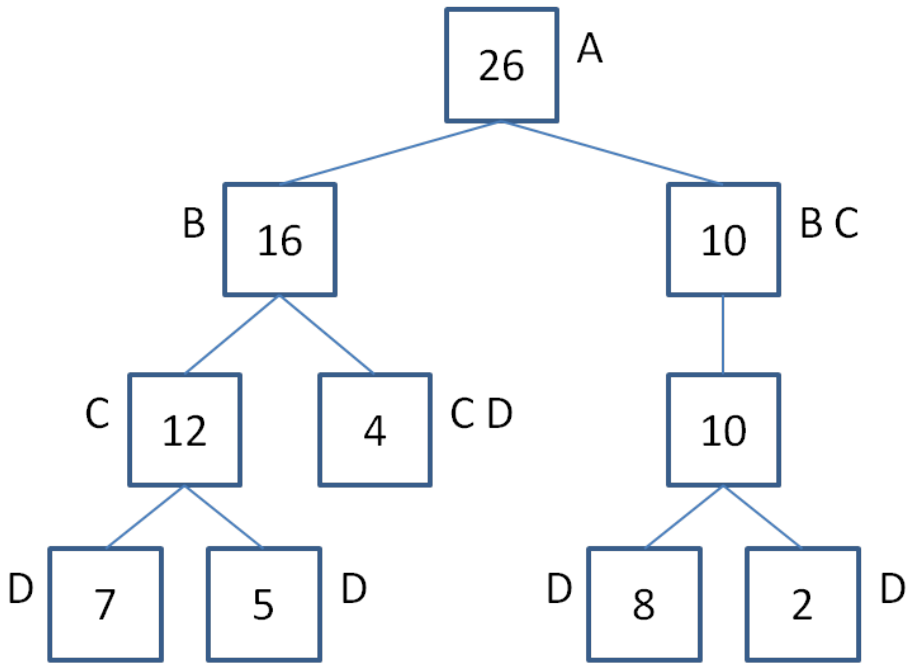

Figure 5.5: Queue Point Selection — Here, we demonstrate how our queue point selection algorithm works on a toy k-d tree. We measure the amount of cache available to hold geometry and determine the maximum amount of geometry (g_{max}) that can be loaded without exceeding available cache capacity. We 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 (marked **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 marked **B** are queue points. If $16 > g_{max} \geq 10$, the **C** nodes are queue points. If $g_{max} \leq 10$, the leaves (marked **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 5.4.3).

5.4.1.1 Traversing Primary Rays

To handle secondary rays in our dynamic scheduler, we found it necessary to adjust our initial algorithm. Rather than targeting only large leaves, we now select queue points in the acceleration structure based on the amount of geometry that fits in available cache. Each queue point is the root of a subtree of the acceleration structure, a subtree that contains as much geometry as possible, but no more than will fit into L2 cache. See Figure 5.5 for an example. If the entire scene fits into cache, then the root of the acceleration structure is the only queue point, and a recursive traversal is used.

Our algorithm efficiently schedules for worst-case conditions. Sometimes the construction of 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 by loading blocks of both rays and geometry to process the queue efficiently.

Our algorithm enqueues 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 processed, 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 might reach. Once a ray is selected at this stage, it is processed until either a successful intersection is found or until the ray exits the bounds of the subtree. If the ray exits the bound of the subtree, it continues its traversal through the full acceleration structure, either

to the next queue point or until it exits 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 super-sampling, samples can be blended in the framebuffer as they arrive. If secondary rays are cast, then the point is shaded iteratively as each secondary ray is processed.

5.4.1.2 Traversing Secondary Rays

Our algorithm traces secondary rays in generations according to ray type: 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. 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 at the queue point that

contains its origin. Our algorithm then iterates over queue points to traverse rays, as before. While our algorithm may achieve less coherence here than for primary and shadow rays, it can achieve better ray-object 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 resulting intersection points are used to generate the next generation of shadow and secondary rays. This process continues until no new secondary rays are generated.

5.4.2 Tiling Ray and Scene Data

While we use a recursive traversal to test each single ray against the required scene data at a queue point, this is not the only strategy available. As we mentioned in Section 2.1.3.3, the localized groups of rays and scene data at each queue point create a dense, regular data layout, and the traversal and intersection iterations resemble the nested loop iterations used in matrix computations. Thus, we can apply loop tiling or blocking optimizations used for dense matrix operations [7, 20, 69, 73, 120] to improve data locality and reuse at the highest levels of the memory hierarchy. Our approach makes these transformations possible by collecting significant numbers of *both* ray and scene data at each queue point and by bounding the total data amount per queue. Note that these optimizations cannot be applied easily to recursive ray tracers because the general recursive traversal contains too many conditional steps to effectively block scene data and because recursive tracers maintain too little active ray state to make blocking worthwhile.

Loop tiling improves data locality and reuse for a nested loop by keeping

a tile of data for one variable X resident in memory while iterating across data for another variable Y. In this way, each tile of X is loaded once and each tile of Y is loaded once for each tile of X. If we define $Cost(X)$ to be the bandwidth cost of loading a tile of X and $Tiles(X)$ to be the number of tiles of X, we can calculate the total cost for a particular tiling strategy with Formula 5.1.

$$Cost(X) \cdot Tiles(X) + Cost(Y) \cdot Tiles(Y) \cdot Tiles(X) \tag{5.1}$$

For ray tracing, we use rays and scene objects as variables, where the scene objects include both acceleration structure data and geometry [3]. Either ray tiles or object tiles can be kept resident (used as the X variable), though the best choice depends on the size of the target memory, the number of tiles of each type, and the bandwidth cost to load each tile type. In Figure 5.6, we show bandwidth costs for both variants across a range of ray and object counts and assuming a 32 KB cache.

5.4.3 Implementation Sketch

We now describe how our algorithm can be implemented on a multi-core processor, assuming a 4MB L2 cache. We keep geometry and acceleration structure data cached while streaming rays to keep threads maximally occupied.

We represent k-d tree nodes using eight bytes, similar to the k-d tree used in PBRT [84]. We use an additional bit from the least-significant end of the mantissa

[3] For generality, we include both acceleration structure data and geometry data, since the acceleration structure tests determine which geometry data is tested. If the queue point is at a leaf of the acceleration structure, then only geometry is included.

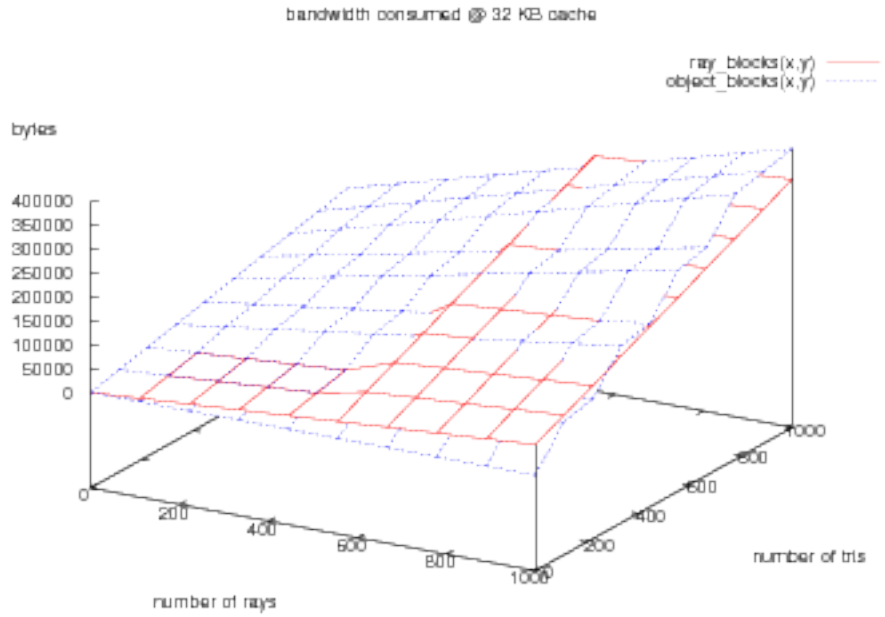

Figure 5.6: Bandwidth cost when tiles of rays or tiles of objects are kept resident for a 32 KB cache assuming 64 bytes per ray and 128 bytes per triangle object. Tiles are sized so that the maximum number of rays or objects can be kept resident at once.

of the split location to indicate whether a node is a queue point, leaving 20 bits for the single-precision mantissa representation. We expect this quantization to not significantly affect the quality of the k-d tree. With this representation, 128K nodes can remain resident if we reserve 1MB of the L2 for nodes. If the k-d tree is larger, we can use a "forest and sub-tree" structure [93] by maintaining only the top of the tree, from the root down to the queue points, then loading each subtree before processing its associated queue. This structure is only needed 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. This table costs 8 bytes per entry, and we expect 32K queue points to be sufficient for most trees, so the table requires 256KB.

Rays must be cached to perform traversals and intersections, yet we expect to have hundreds to thousands of rays queued at each point. For example, if the camera point lies within the acceleration structure, all primary rays will begin in one queue. If we bring in too many rays, we evict other needed data from cache, so we buffer only 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 as long as there are queued rays. Each traversal step in a k-d tree computes a ray-plane intersection test, which is 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 [100]. Assuming current DRAM latencies and an average of ten traversal steps to take a ray from the queue point to a leaf, two to four rays per thread should be sufficient to prevent idle threads.

We use a tight, 64-byte ray representation, with support for adaptive sampling. Our ray layout supports up to 160-bit color, which must be included with each ray for linear processing. We include space for a pointer to adaptive sampling information, if required. We pad the structure to 64 bytes to maintain cache alignment.

Finally, we must 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 cannot know which geometry, if any, will actually be tested for intersection. To avoid spurious geometry loads, we lazily load geometry when a ray needs it for an intersection test. If a queue point is at a leaf, then we load the geometry immediately to be tested for intersection.

5.5 Experimental Methodology

This section describes how we obtained our results. We perform all experiments using an unoptimized ray tracer with a simulated L2 cache. We evaluate the performance of our algorithm in terms of cache utilization and bandwidth consumption.

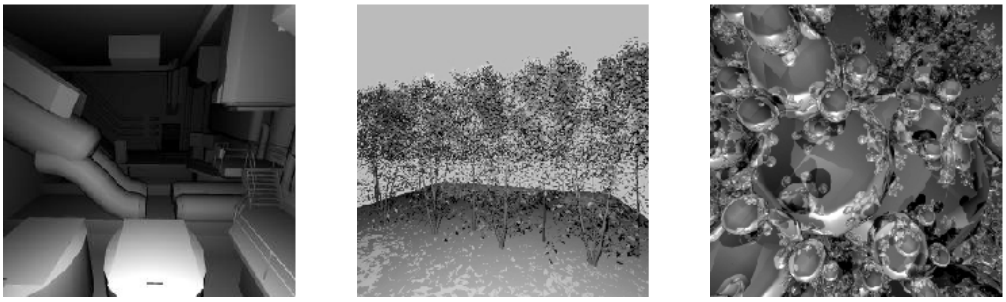

Figure 5.7: Test scene images — We test our complete algorithm on three scenes: **room**, **grove**, and **sphereflake**. Note that the geometry artifacts in **room** are contained in the scene specification and are not due to our ray tracer.

To obtain the cache measurements for our simulation, we create a memory trace with explicit reads and writes, then run that trace through various cache configurations using Dinero IV [31], a light-weight trace-driven simulator. We simulate cache sizes in power-of-two increments from 1KB to 4MB, with 64B cache lines. This size range allows us to measure the performance of our algorithm both when cache resources are scarce and when they are plentiful. Each cache is fully associative to eliminate conflict misses. Thus, after the cache is warm, all misses are capacity misses.

Our simulator is single-threaded, which eliminates multi-threaded cache issues, such as false sharing, from our measurements. In an optimized implementation, threads will collaboratively process the rays at a single queue point. Because the geometry associated with each queue point fits in available cache, we expect to avoid most thread-related contention.

We test our algorithm on three scenes, shown in Figure 5.7. The scenes are rendered at 1024×1024 resolution for all runs except for diffuse reflections, where they are rendered at 512×512 resolution. These models provide a variety of total geometry, potentially-visible geometry, and geometric topology. We use **room**, a small architectural scene (47K triangles); **grove**, a grove of tree models (164K triangles); and **sphereflake**, a four-level sphereflake model from the Standard Procedural Database [38] (797K triangles). We specifically mention potentially-visible geometry when tracing primary and secondary rays because it more accurately measures the geometry loaded 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, which is an ordering known to produce good image-plane coherence [104]; and a ray packet tracer using 8×8 packets tiled over the image plane. We use an 8×8 packet size since it is the midpoint of currently popular packet sizes (4×4 [91, 110] to 16×16 [109]) and it was recently found to provide the most speed-up among sizes in this range [17].

5.6 Results and Discussion

This section shows that dynamic ray scheduling can improve cache utilization for geometry data by as much as an order of magnitude over recursive ray tracing. These savings will become increasingly important for rendering complex

models that require more data to be loaded, adding additional stress on the cache.

5.6.1 Interpreting the Result Charts and Tables

Figures 5.8 – 5.13 provide a comparison of the bandwidth consumed by our approach to the bandwidth consumed by the single-ray recursive tracer. Tables B.1 – B.12 in Appendix B show all our results, including the bandwidth consumed when tracing 8×8 packets. The tables show both the raw bandwidth measurements and the relative improvement of each approach compared to the recursive ray tracer performance.

For each configuration, we list the amount of geometry that is *potentially visible*. This is the amount of geometry that is tested for intersection against at least one ray, and thus it is the total amount of geometry that must be loaded into the cache. This amount represents the cache pressure from geometry more accurately than either total geometry or visible geometry, since total geometry includes geometry that is culled by the acceleration structure and visible geometry omits geometry that is tested for intersection but either is not hit or is occluded by other nearby geometry.

Each table includes both the size of the cache sizes tested and the fraction of the potentially visible geometry that fits into each cache size. This fraction expresses the cache pressure in the system for a particular combination of scene and cache size. Smaller values indicate greater pressure. Values greater than 1.0 indicate that all potentially visible geometry can be cached, which minimizes cache pressure. This fraction allows a fair comparison among scenes with different

amounts of potentially visible geometry. Cache sizes that hold similar fractions of geometry should be compared (for example, 32K for **room** and 512K for **grove**), rather than comparing results for the same cache sizes.

When tracing shadow rays, the amount of potentially visible geometry differs slightly for our algorithm and the other algorithms. This is because shadow rays in our algorithm are traced from the light source to the non-shadow hit point, whereas the other algorithms tested trace from hit point toward the light source.

5.6.2 Tracing Primary Rays

We present our measurements for tracing primary rays in Figures 5.8, 5.10 and 5.12. These results 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. In other words, we want to keep geometry in cache and stream rays, as 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 thrashes the cache with geometry data. The problem is 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

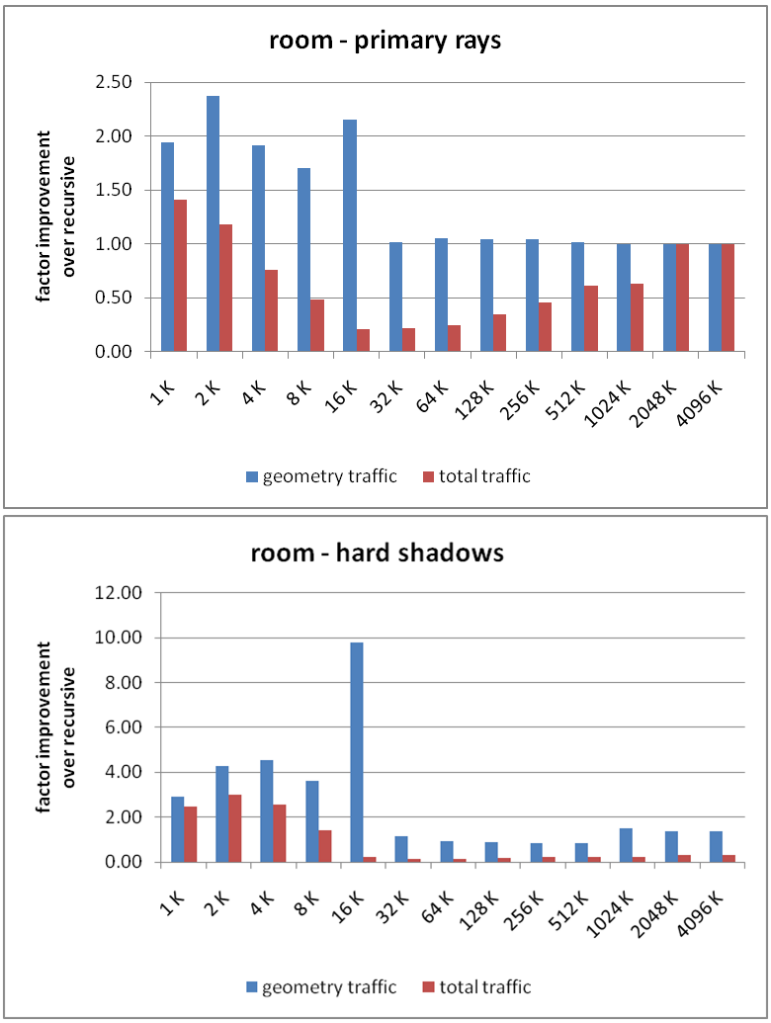

Figure 5.8: Performance improvement for dynamic scheduling over recursive tracing for primary rays and for hard shadows on **room** — Numbers greater than one indicate better bandwidth utilization compared to single-ray recursive tracing. See Tables B.1 and B.2 for the underlying data.

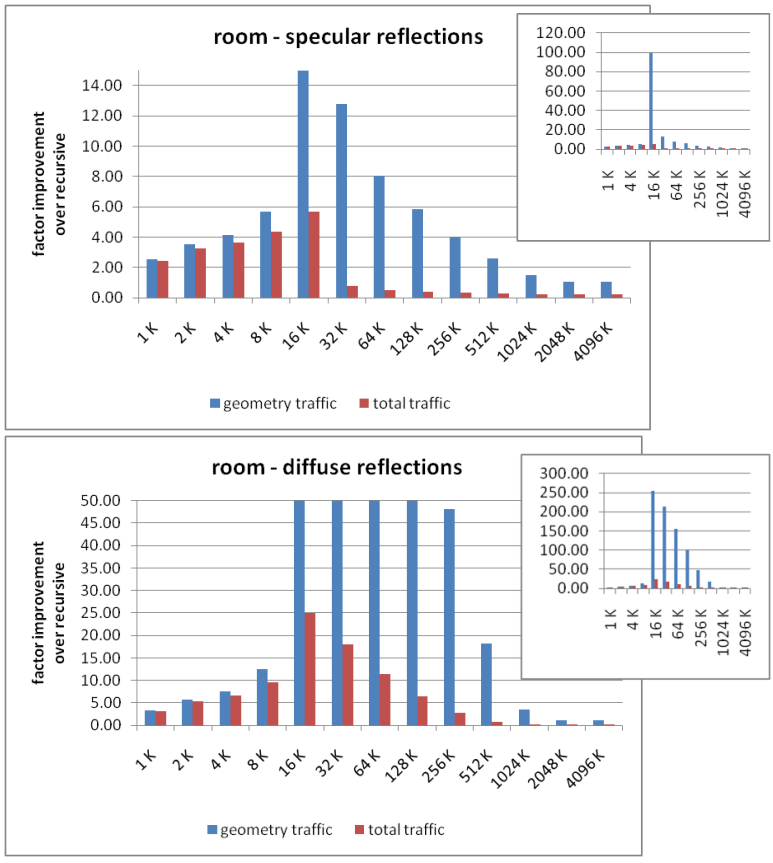

Figure 5.9: Performance improvement for dynamic scheduling over recursive tracing for specular reflections and for diffuse reflections on **room** — Numbers greater than one indicate better bandwidth utilization compared to single-ray recursive tracing. See Tables B.3 and B.4 for the underlying data.

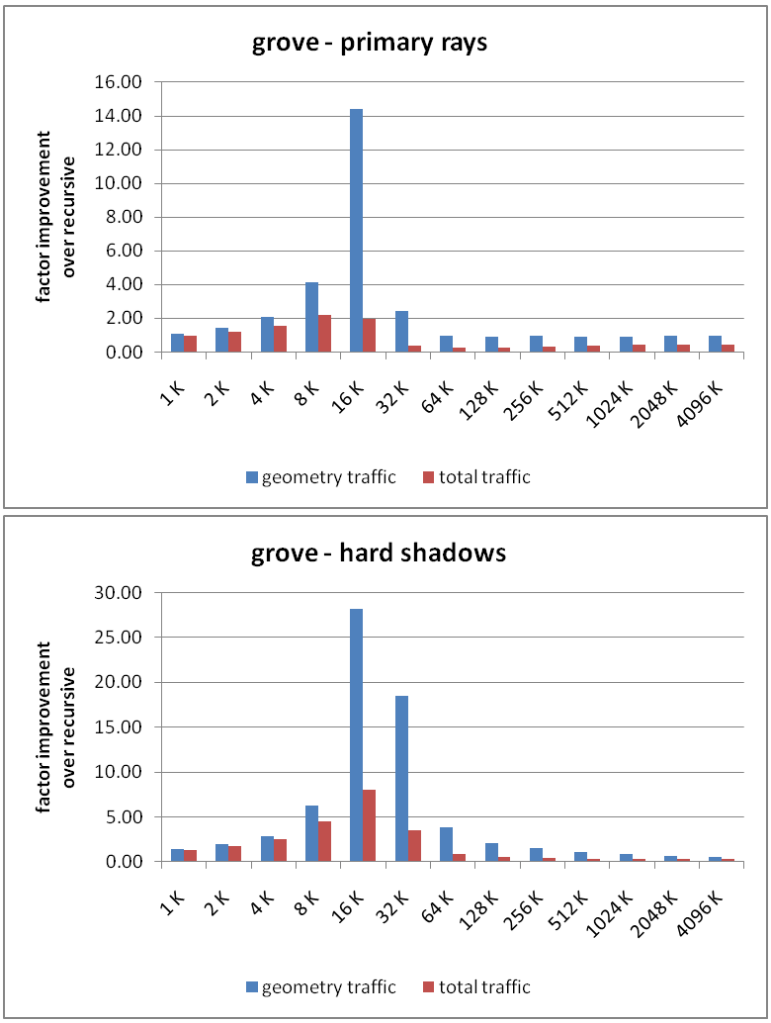

Figure 5.10: Performance improvement for dynamic scheduling over recursive tracing for primary rays and for hard shadows on **grove** — Numbers greater than one indicate better bandwidth utilization compared to single-ray recursive tracing. See Tables B.5 and B.6 for the underlying data.

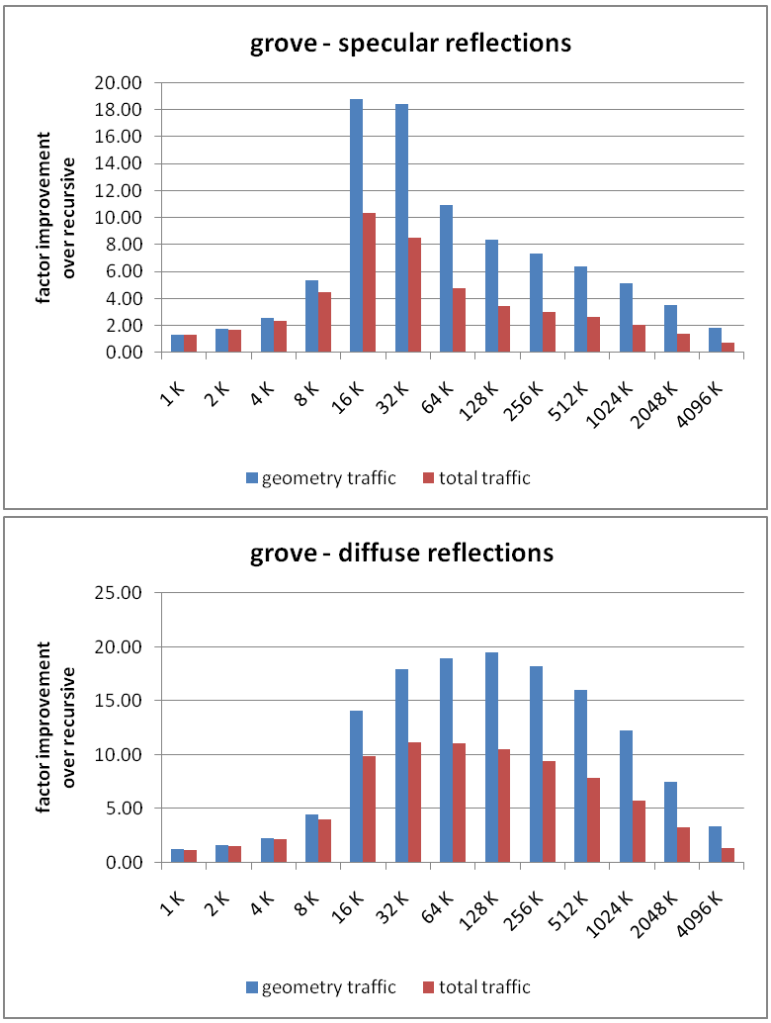

Figure 5.11: Performance improvement for dynamic scheduling over recursive tracing for specular reflections and for diffuse reflections on **grove** — Numbers greater than one indicate better bandwidth utilization compared to single-ray recursive tracing. See Tables B.7 and B.8 for the underlying data.

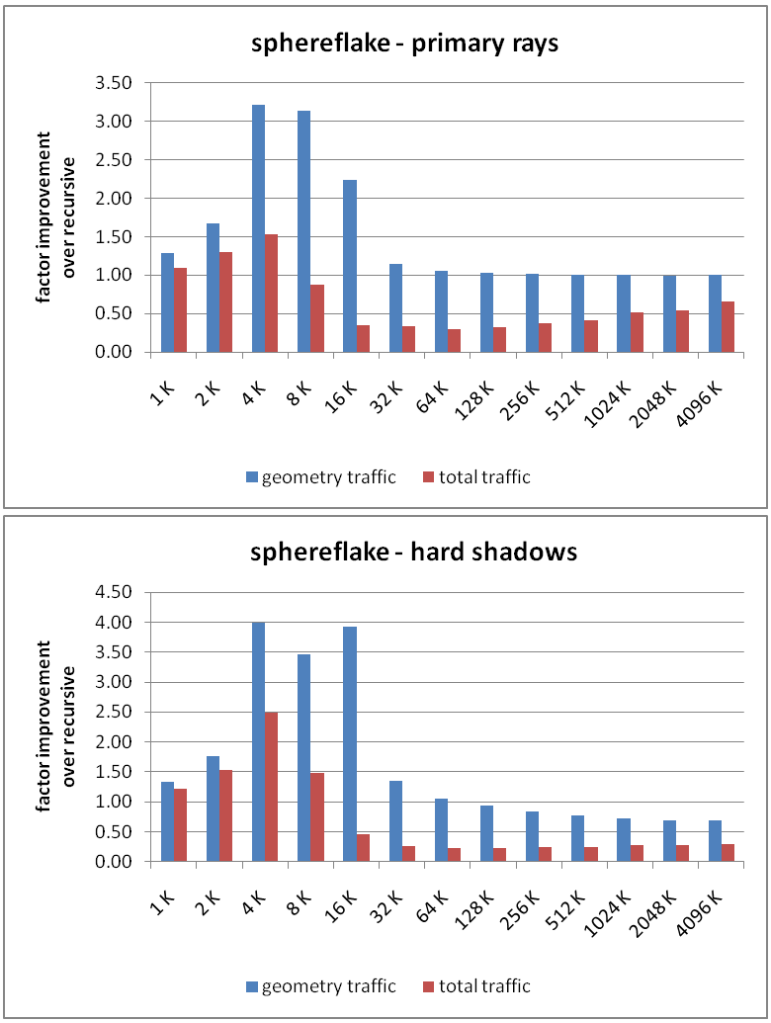

Figure 5.12: Performance improvement for dynamic scheduling over recursive tracing for primary rays and for hard shadows on **sphereflake** — Numbers greater than one indicate better bandwidth utilization compared to single-ray recursive tracing. See Tables B.9 and B.10 for the underlying data.

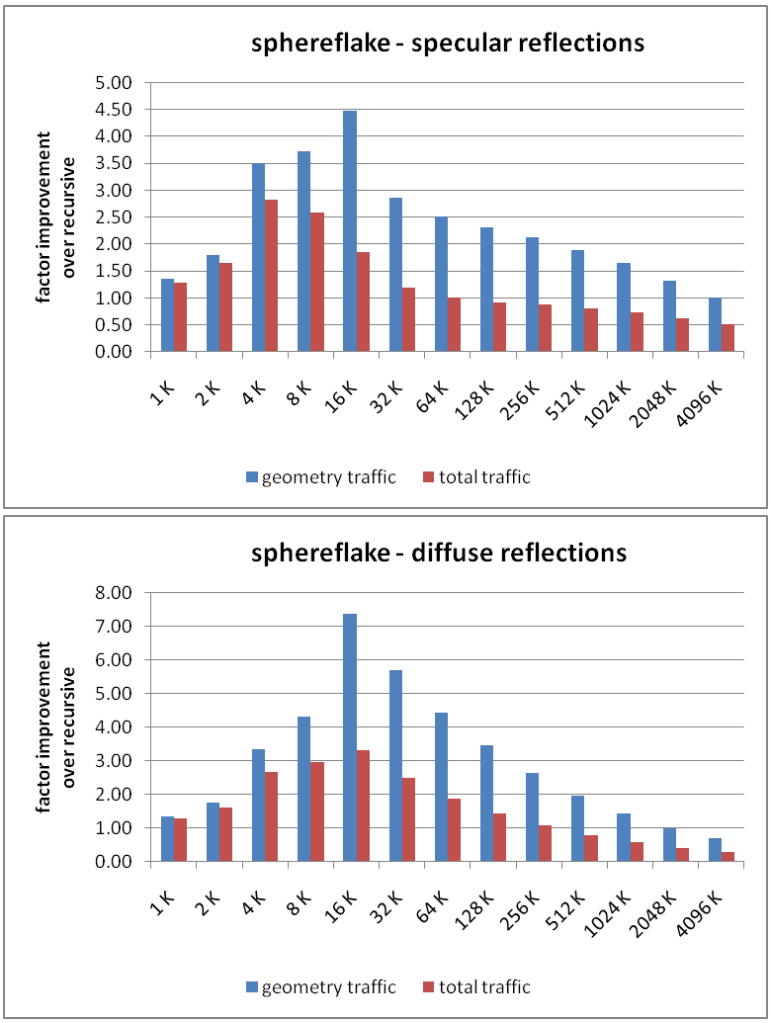

Figure 5.13: Performance improvement for dynamic scheduling over recursive tracing for specular reflections and for diffuse reflections on **sphereflake** — Numbers greater than one indicate better bandwidth utilization compared to single-ray recursive tracing. See Tables B.11 and B.12 for the underlying data.

can make efficient use of system resources and adapt to various system loads.

5.6.3 Tracing Secondary Rays

Our algorithm performs well when both primary and secondary rays are traced. We present measurements for primary rays, shadow rays from three point-lights, specular reflections, and diffuse reflections. We separate the specular and diffuse reflection cases to better observe their respective behavior. All reflections are limited to two bounces.

We use hemispherical sampling to generate diffuse reflection rays, similar to techniques used in Monte-Carlo global illumination techniques and in photon-mapping [53]. Nine reflection rays are generated per sample, using a 3×3 sampling grid over the hemisphere. Each bounce has only a 10% chance of generating another sampling round. Our sampling is insufficient for accurate global illumination simulation, but it provides a reasonable approximation of system performance when tracing maximally incoherent rays.

In Figures 5.8 – 5.13, we present the traffic from main memory to processor cache for our algorithm when tracing hard shadows and incoherent reflection rays (both specular and diffuse). The relative performance of our algorithm *increases* as ray incoherence and ray density increase, and the total data traffic of our algorithm never explodes as it does in the recursive algorithms tested.

Our algorithm performs best when visible geometry is large compared to available cache and when many rays are active at once. For example, on the 164K triangle **grove** scene rendered with hard shadows and diffuse reflections with

512KB cache available, our algorithm reduces DRAM-to-L2 bandwidth consumption 7.8× compared to packet ray tracing. In contrast, on the 797K triangle **sphereflake** scene with the same configuration, our algorithm does not reduce bandwidth consumption because the scene is over-tessellated for the sampling rate used. Each sphere is tessellated fully in the scene representation, which can be unnecessarily fine for a sphere that covers only a few pixels and against which few rays are tested for intersection. For such scenes, a multi-resolution system similar to Djeu et al. [30], with coarsification added, would boost our algorithm's performance.

Our algorithm does comparatively worse when the working set of the recursive algorithm fits in available cache. However, we believe these negative results will not translate into negative system performance because they only occur when much of the scene fits in cache. In such cases, the additional bandwidth consumed under our algorithm occurs when bandwidth is not fully utilized. Further, recursive algorithms struggle to maintain memory locality under complex lighting models. Global illumination approximations can 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.

5.6.4 Results for room

Tables B.1 – B.4 contain the measured traffic from main memory to processor cache 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 smaller tested cache sizes. For diffuse reflection rays, when rays are maximally incoherent, we can reduce geometry traffic by as much as *two orders of magnitude* (16K – 128K) and total traffic by as much as an order of magnitude. When the working set for the recursive algorithm fits in available cache, the relative performance of our algorithm suffers because of the overhead to maintain the ray queues. Note that while our relative performance numbers are worse for these resource-plentiful situations, the total bandwidth consumed is relatively low, so the impact on system performance is small.

5.6.5 Results for grove

Tables B.5 – B.8 contain the measured traffic from main memory to processor cache for **grove**.On this larger test scene, the benefit of our algorithm becomes clear. When cache resources are scarce, our algorithm significantly reduces data traffic. When cache resources are plentiful our algorithm achieves better cache utilization with respect to geometry, at the cost of increased ray traffic. The results for incoherent rays (specular and diffuse reflections) are particularly striking, where our algorithm significantly reduces total data traffic even with megabytes of available cache.

5.6.6 Results for sphereflake

Tables B.9 – B.12 contain the measured traffic from main memory to processor cache for **sphereflake**. This scene has both the most and the most potentially-visible geometry in our test set. Although our performance here is worse than on

other test scenes, the bandwidth usage is relatively low in all cases, so we expect the overhead of our algorithm to have little impact on system performance. This scene is a worst case for our algorithm because the geometry is severely over-tessellated: The smallest spheres are fully represented, which concentrates tens-of-thousands of triangles in the area of a few pixels. As a result, there is very little geometry shared among ray intersections. Also, the queue-points for our algorithm are deep in the k-d tree, which limits the number of rays queued and thus the coherence benefits of our algorithm. As discussed in Section 5.6.3, a multi-resolution geometry representation would boost our relative performance for this case.

5.7 Summary

In this chapter, we presented a ray tracing algorithm that dynamically enqueues and schedules rays in order to actively manage both ray and geometry data. Compared to recursive traversals, our algorithm significantly reduces single-processor DRAM-to-L2 geometry traffic and moderately increases ray traffic. Our algorithm dramatically reduces total bandwidth consumed between main memory and processor cache when cache resources are scarce, which makes it particularly suited for ray tracing large, complex scenes. When cache resources are plentiful, our algorithm can consume more bandwidth than recursive algorithms due to increased ray traffic, though the absolute bandwidth consumed in such cases remains small.

We have shown that our dynamic algorithm can efficiently manage memory for single core, and previous work [85] shows that queueing tracers can manage

memory for a single machine. Therefore, we expect our algorithm to be well-suited for ray tracing scientific data on large clusters, which are the dominant architecture in modern supercomputing [103]. These clusters generate datasets of increasing size and complexity, the largest of which cannot be efficiently moved from the originating machine. There is a growing need for rendering algorithms that can operate efficiently on the distributed architectures where such large data is produced. In Chapter 6, we develop a dynamic scheduling algorithm to efficiently render large, complex scenes that require the assembled resources of many distributed machines.

Chapter 6

Dynamically Scheduled Distributed Memory Ray Tracing

Our work to this point demonstrates that dynamic scheduling is particularly well-suited for rendering tasks that operate on large datasets under tight memory constraints. This suggests that a dynamic scheduling algorithm might be a good candidate to render large-scale scientific data on large distributed clusters. This rendering problem is particularly challenging because each dataset can be as large as the aggregate memory of tens or hundreds of cluster nodes and each individual node has a relatively small amount of local RAM per core. A suitable rendering algorithm must scale to hundreds of nodes in order to gather enough aggregate memory for the largest datasets, and it must function within the limited per-core resources at each node. In this chapter, we explain how the dynamic scheduling algorithm can operate on large distributed clusters.

6.1 Overview

In Chapters 3 and 4, we showed that recursive ray tracing operates efficiently when all required data can fit in memory. This result remains true for large-scale parallel ray tracing. Many previous large-scale recursive ray tracers assume they

operate in shared memory, a single global memory address space across the entire machine. This assumption is attractive because such memory can be used like the memory on a single-core machine, just at a larger scale. As in the single-core case, however, these algorithms struggle if required data is larger than available memory. In addition, large-scale shared memory environments typically have non-uniform access times across the address space; the environment might not indicate that a memory request will have extra latency, which complicates handling such requests efficiently.

Most large scientific simulations are now run on distributed memory supercomputing clusters [103] for which shared memory assumptions do not directly apply. Shared memory can be simulated as a virtualization layer between the application and the hardware [26], but such a layer can exacerbate the non-uniform access penalty for requests for non-local memory. Further, the layer does not change the memory request pattern for a ray tracer: even if the shared memory system caches non-local data [26], we have shown that the many incoherent memory requests generated by recursive tracing will result in poor memory performance.

To further complicate the situation, many large simulations produce terabytes of data per timestep. Such data are too large to move off the machine that produced them to a different machine for processing. Therefore, we assume that a ray tracer must run on the same distributed-memory machine that produced the data. Further, it is impractical, and sometimes impossible, to pre-compute highly-tuned acceleration structures for these large datasets. Pre-processing requires significant additional machine-time and disk space, and the resulting acceleration structure

would consume significant additional DRAM during rendering which can exceed the available memory. It is not clear that researchers will be willing or able to spend part of a finite machine allocation to generate these structures, particularly if the data will not be rendered enough times to offset the cost.

In this chapter, we present a dynamic ray scheduling algorithm for distributed-memory ray tracing. Our algorithm schedules ray traversal and intersection calculations according to the data resident on each processor in a manner similar to scheduling for a single core. By associating rays and data into locally coherent work units, our algorithm can adjust to evolving data requirements and alter the schedule of these work units across the parallel environment to achieve better overall system performance. We show that dynamic scheduling can improve performance for large datasets where disk I/O limits performance. Our algorithm is able to render a 650 GB *n*-body dataset on a 1024 core cluster, which a statically scheduled ray tracer could not. For a smaller n-body data set for which the static schedule does complete, our dynamic scheduler reduces data loads by $10\times$ to $48\times$. Our scheduler also demonstrates better performance than static strategies for volumetric ray casting.

The remainder of this chapter proceeds as follows. We first discuss related large-scale ray tracing work in Section 6.2. In Section 6.3, we present our approach in detail. We describe our testing methodology in Section 6.5, including a light-weight event based simulator we built to test schedules prior to full implementation. We present our full system results in Section 6.6; the raw data from our results are in Appendix C.

6.2 Related Work

In this section, we place our approach in the context of prior work and other approaches to large-scale ray tracing.

6.2.1 Shared-Memory Ray Tracing

Most recent ray tracing systems use algorithms designed for shared memory architectures, including multi-core workstations and larger symmetric multiprocessors (SMPs). Significant shared-memory systems include RTRT (also called *-Ray) by Parker et al. [78, 79, 80], Reshetov et al.'s multi-level ray tracing algorithm (ML-RTA) [91], and SCI's Manta ray tracer [16]. Wald et al. [112, 113] provide a deeper discussion of the current state of the art for shared memory approaches.

6.2.2 Distributed-Memory Ray Tracing

A number of specialized, distributed-memory ray tracing architectures were proposed though the 1980s, though few were actually manufactured and most performance results were obtained through simulation. Green and Paddon [36] survey these systems in detail.

We are aware of several distributed-memory ray tracers that divide data across processors and send each ray to the processor that contains its required data. Each of these systems use specialized hardware rather than a cluster of general-purpose machines. Dippé and Swensen [29] proposed a specialized architecture for distributed-memory ray tracing that loads scene data across aggregate system memory, then dynamically changes the acceleration structure boundaries to load-

balance the system. This requires frequent data communication among the affected processors in addition to ray traffic. Kobayashi et al. [59] statically subdivide scene space among processors and use a number of processing elements at each processor to perform ray calculations. This approach may map well to modern clusters, but overhead from small and frequent ray communication would likely degrade performance. Dachille and Kaufman [25] used ray reordering and queueing in a specialized hardware-based volume renderer. They support lighting, including simple global illumination and scattering effects, and reflections. They schedule rays using a static assignment of spatial subdivisions to processors, and pass rays among them.

Recent non-queueing distributed-memory ray tracers divide work according to image coherence, either by tracing contiguous pixels to achieve coherent scene references or by tracing disparate pixels to achieve better load balance. Salmon and Goldsmith [93] compare these two approaches. Green and Paddon [36] proposed, to our knowledge, the first distributed-memory ray tracer using general-purpose machines. Their approach uses an image-space distribution of rays with load balancing achieved by serving rays on-demand to processes that require work. They use a initial low-resolution rendering pass to load required data on processors, and each processor receives a contiguous block of pixels based on the initial low-resolution pass. Wald et al. [107, 114] trace disparate pixels to load-balance their system and achieve interactive rates with a ray tracer on a small cluster, but this technique requires significant pre-processing of the dataset to achieve better data coherency. DeMarle et al. [26, 27, 28] adapt *-Ray to run on a distributed memory cluster at

interactive rates, though they also rely on preprocessing the data to achieve better data coherency. They also statically distribute scene data to processors, then rely on a distributed shared memory subsystem to cache required data not stored locally. Such a system requires excess memory per node for this cache, and system performance degrades significantly if caching memory is unavailable.

Scherson and Caspary [94] used a data distribution, but load-balanced tasks between ray traversal and ray intersection tasks. Lefer [64] introduced a hybrid scheduling scheme for distributed memory ray tracing, where data is distributed across the cluster and ray tasks are pulled on-demand to processors that need work. Reinhard et al. [89, 90] implement another hybrid approach that combines data distribution across the cluster with tasks assigned to processes based on load. This approach keeps camera and shadow rays on the originating process, while passing reflection and refraction rays to a process that contains the data required to process them. The Kilauea system [57, 58] maps the entire scene space across all processors, but distributes only part of the scene data to each to achieve load balancing and scalability. Each ray is duplicated and sent to all processes. After all copies of a ray have been traced, the originating process collects the results and determines the front-most hit point. Scaling results are reported only to sixteen processes. This system requires scene data to fit entirely in aggregate memory, and it is unclear how its small, frequent ray communication will scale beyond tens of processes. It is also unclear whether the system can accommodate scientific data that does not have pre-tessellated surfaces.

6.2.3 Large-Scale Direct Volume Ray Casting

Recent work in large-scale ray casting uses a fixed data decomposition approach to render images across thousands to hundreds-of-thousands of processors [22, 44, 82]. These approaches process the entire dataset in-core by rendering each sub-domain separately and then composite the results into the final image. This approach, which ultimately is a form of speculative execution, is effective because there is a fixed and regular amount of work to perform in the absence of reflections. However, when including reflections, this fixed data approach is prone to load imbalance. Further, the speculative rendering of these approaches wastes the work used to generate sub-domain images that are rejected by the final image composition pass.

6.3 Algorithm Overview

In this section, we summarize ray scheduling for distributed memory machines and discuss the potential advantages of our approach.

6.3.1 Distributed-Memory Ray Scheduling

In Chapter 5, we demonstrated that a dynamically scheduled queueing ray tracer can improve memory efficiency for single core rendering. We also discussed in Section 4.5 how Pharr et al. [85] showed that queueing can improve efficiency between disk and system RAM. We now expand the scope of our dynamic scheduling work to address the I/O bottleneck experienced by previous distributed-memory ray tracers. Using dynamic ray scheduling, we create a distributed-memory system

where I/O costs can be effectively controlled, even when processing incoherent secondary rays. This I/O savings is realized by using a scheduling algorithm that is sensitive to the computational load and the size of the image that is being rendered. We describe our tested schedules below, and give pseudocode for each algorithm in Figures 6.2 – 6.4.

We establish a performance baseline for our approach by using two static schedules inspired by previous systems, though our implementation varies in this important respect: our queueing implementation builds more coherence than the non-queueing implementations referenced below, so we expect better memory system performance in our implementation, especially for our largest out-of-core tests. Since none of these systems enqueue rays, we expect this schedule to be a conservative comparison against our dynamic schedules. The I/O behavior of this schedule will be more favorable than it would be in the recursive framework used by the related work.

Image Decomposition — rays are evenly divided across processes by contiguous image plane decomposition, and data is loaded as ray computation requires. At each scheduling step, each process selects the domain with the most local rays queued. This is most similar to previous image coherence strategies employed [16, 26, 27, 28, 36, 79, 80, 107, 114]. This schedule also corresponds to the demand-driven component of the schedule used by Reinhard et al. [89, 90]; and it represents the extension of Pharr et al.'s single-machine queueing algorithm [85] to a distributed-memory machine, though we do not

consider the final image contribution of each ray. See pseudocode in Figure 6.2.

Domain Decomposition — data is partitioned into spatial domains then domains are evenly divided across processes. A process can be assigned multiple domains if there are more domains than processes, or it can be assigned no domain if there are more processes than domains. Note that domain data is loaded at first use, rather than prefetched. Rays are sent to process that contains data needed for computation. At each scheduling step, each process selects the assigned domain with the most local rays queued. This is similar to previous domain decomposition strategies [25, 29, 59]; and to the data parallel component of the scheduling strategy in Reinhard et al. [89, 90]. This also corresponds to the data distribution used in large-scale volume renderers [22, 44, 82]. See pseudocode in Figure 6.3.

We test two dynamic ray schedules to compare their potential benefit against the static schedule baseline. The two dynamic schedules are identical except for whether the number of waiting rays are considered. By this, we can verify the importance of including the number of rays in the scheduling metric. See pseudocode in Figure 6.4.

Spread — rays are evenly divided across processes. After the initial ray distribution, each scheduling step sends rays to processors that already contain the domain data required for intersection. Processors that have domain data not

```
ProcessQueue(queue)
{
  while (! queue.empty() )
  {
    r = q.top();
    q.pop();
    // intersection for ray tracing
    // traversal for direct volume ray casting
    // generated rays inserted into q
    PerformRayOperations(d, r, q);
    if (! RayFinished(r) )  Enqueue(queues, r);
    else ColorFramebuffer(r);
  }
}
```

Figure 6.1: Pseudocode for ProcessQueue(), used in each schedule pseudocode (see Figures 6.2 – 6.4).

```
ImageDecompositionTrace()
{
  // produces ray division for the current proc
  rays = GenerateRays();
  // generates an ordered list of domains for each ray
  // enqueues them for first domain each requires
  queues = EnqueueRays(rays);
  while (! queues.empty() )
  {
    q = FindQueueWithMostRays(queues);
    d = LoadDomain(q.domain_id);
    ProcessQueue(q);
    queues.delete(q);
  }
  MergeFramebuffers();
}
```

Figure 6.2: Pseudocode for Image Decomposition Static Schedule.

```
DomainDecompositionTrace()
{
  // produces all camera rays
  rays = GenerateRays();
  // generates an ordered list of domains for each ray
  // enqueues rays that require a domain assigned to this process
  // discard other rays (they will be enqueued by another process)
  queues = EnqueueRays(rays);
  last_d = NONE;
  done = FALSE;
  while (! done )
  {
    q = FindQueueWithMostRays(queues);
    if (q.domain_id != last_d)
    {
      d = LoadDomain(q.domain_id);
      last_d = q.domain_id;
    }
    ProcessQueue(q);
    queues.delete(q);
    // for each queue where
    // the domain is assigned to another process
    // send those rays to the assigned process
    // receive rays coming here
    SendRaysToNeighbors(queues);
    done = NoProcessHasRays();
  }
  MergeFramebuffers();
}
```

Figure 6.3: Pseudocode for Domain Decomposition Static Schedule.

```
DynamicScheduledTrace()
{
  // produces ray division for the current proc
  rays = GenerateRays();
  // generates an ordered list of domains for each ray
  // enqueues them for first domain each requires
  queues = EnqueueRays(rays);
  q = FindQueueWithMostRays(queues);
  d = LoadDomain(q.domain_id);
  last_d = q.domain_id;
  done = FALSE;
  while (! done )
  {
    if (q.domain_id != last_d)
    {
      d = LoadDomain(q.domain_id);
      last_d = q.domain_id;
    }
    ProcessQueue(q);
    queues.delete(q);
    // determine schedule for next round
    // send rays to assigned processors
    // receive rays coming here
    q = ScheduleNextRound(last_d, queues);
    done = NoProcessHasRays();
  }
  MergeFramebuffers();
}
```

Figure 6.4: Pseudocode for Dynamic Schedules. ProcessQueue() is defined in Figure 6.1 and ScheduleNextRound() is defined in Figure 6.5.

```
ScheduleNextRound(loaded_domain, queues)
{
  foreach q in queues
      queue_info.insert( pair(q.domain_id, q.size()) );
  SendQueueInfoToMaster(loaded_domain, queue_info);
  if (isMaster())
  {
    ReceiveQueueInfo(loaded_domains, queue_infos);
    foreach p in ProcessCount()
        foreach q in queue_infos[p]
        {
          to_schedule.insert(q.domain_id);
          is_loaded.insert( pair(loaded_domains[p], p) );
        }

    foreach p in ProcessCount()
        if (! to_schedule.contains(loaded_domains[p])
            to_evict.insert( p );  // data not needed this round

    foreach domain_id in to_schedule
        if ( is_loaded.contains(domain_id) )
        {
          proc_id = is_loaded[domain_id];
          schedule.insert( pair(proc_id, domain_id) );
        }
        else
            to_assign.insert(domain_id);
    if (UseRayWeighting()) sort( to_assign, queue_infos );
    while (! (to_assign.empty() || to_evict.empty()) )
    {
      domain_id = to_assign.top();
      proc_id = to_evict.top();
      schedule.insert( pair(proc_id, domain_id) );
      to_assign.pop();
      to_evict.pop();
    }
    SendScheduleToAll( schedule );
  }
  ReceiveSchedule( schedule );
  return schedule[ MyProcessId() ];
}
```

Figure 6.5: Pseudocode for ScheduleNextRound(), called for dynamic schedules.

needed by any rays may be assigned to load new domain data that is immediately needed by current rays.

Ray-Weighted Spread — the Spread algorithm, modified so that domains with the most queued rays are preferred when new domain data is assigned to processors. While this schedule considers total rays in a similar way to the cost-benefit analysis used to schedule rays by Pharr et al. [85], it differs in two important aspects. First, because we keep only one sub-domain in memory per process, our schedule acts to keep already-loaded data alive as long as there are rays that require it. Second, we do not include each ray's contribution to the final image as part of our weighting.

Compared to previous work, our approach can render a large scale dataset out-of-core with efficient I/O utilization. We accomplish this by reordering rays and deferring their computation until the required data has been loaded. By dynamically scheduling the queued rays, our algorithm can reduce the number domain loads by over an order of magnitude compared to a static image-plane decomposition and it can achieve better load balancing than a static domain decomposition. We verify this claim with a light-weight ray scheduling simulator, described in Section 6.4, and with a full implementation, described in Sections 6.5 and 6.6.

6.4 Ray Scheduling Simulator

To test the theoretical performance of our schedules, we built a light-weight, event-based distributed-memory ray tracing simulator that measures theoretical per-

Schedule	I/O Behavior	Network Behavior
Image	domains loaded redundantly	no communication across processes
Domain	domains loaded by one process	rays sent to process that contains required data
Spread	domains loaded by one process	rays sent to process that contains required data
Ray-Weighted Spread	domains loaded by one process	rays sent to process that contains required data

Table 6.1: Sketch of the expected schedule behavior on a cluster's network and file system.

	N-Body	CT scan
Comparative	512^3 resolution 512 domains 8.7 GB total size	512^3 resolution 125 domains 1.9 GB total size
Scaling	6144^3 resolution 4096 domains 650 GB total size	4096^3 resolution 4096 domains 116 GB total size

Table 6.2: Dataset Sizes and Decomposition

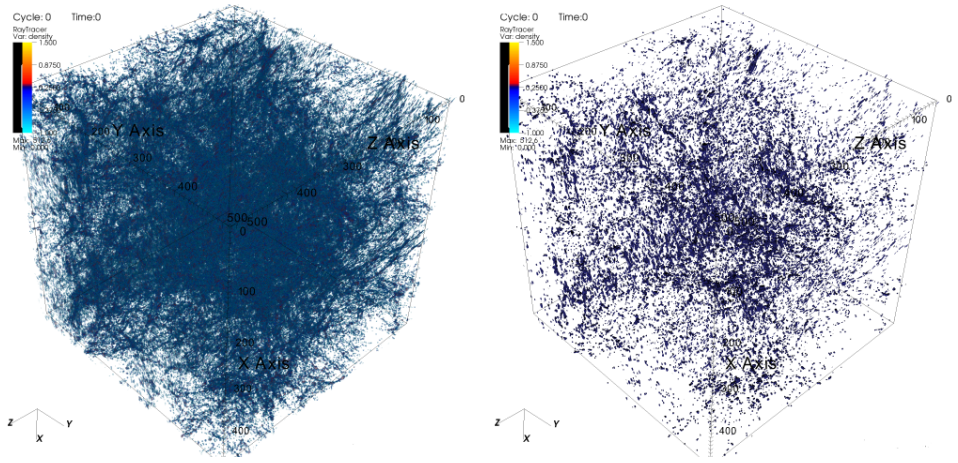

Figure 6.6: Sample images of the cosmology dataset used in our experiments. On left: direct volume ray casting; on right: ray traced isosurface. N-body data courtesy of Ilian Iliev (U. Sussex) and Paul Shapiro (UT-Austin).

formance for given costs of I/O, network communication, and ray traversal and intersection. We tested the image decomposition, domain decomposition and dynamic spread schedules that we described in Section 6.3.1. Our simulator results suggest that when I/O costs dominate the calculation, the dynamic spread schedule reduces bandwidth consumption more than 10× over static scheduling. We found that the results of our full implementation, which we present in Sections 6.5 and 6.6, are in line with the simulator results.

6.4.1 Simulator Description

Our ray tracing simulator is built for rapid prototyping, so it provides reliable quantitative performance estimates without implementing a fully functional tracer. To achieve this, we employ a statistical model of ray behavior: each ray

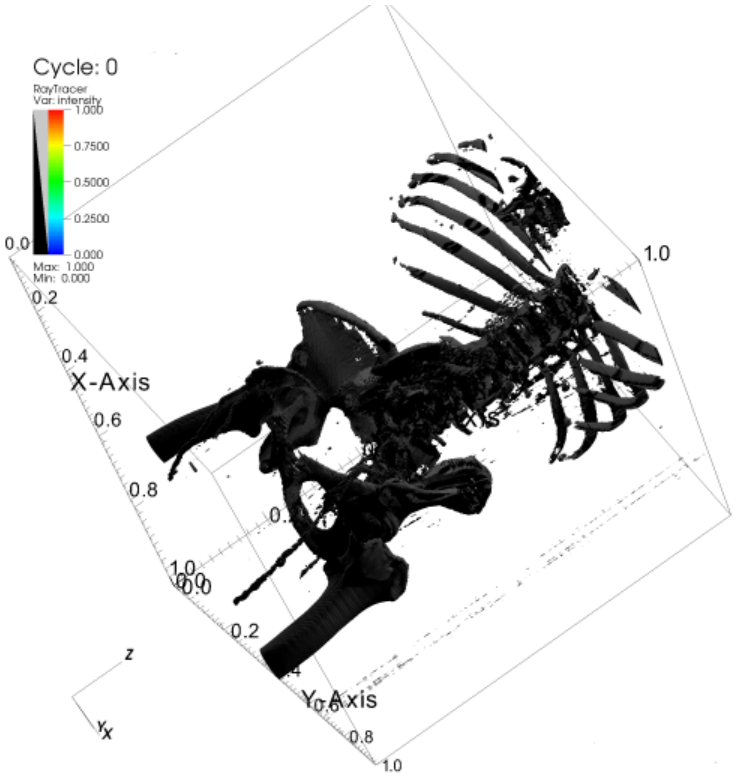

Figure 6.7: Sample image of the CT scan dataset used in our experiments. Image rendered using ray traced isosurface. CT scan data courtesy of Michael Meißner (Viatronix, Inc.)

contains an ordered list of data domains that fall along its path, and each ray performs randomly determined amount of "work" at each domain to simulate traversal and intersection tests. Each ray has a user-defined likelihood of terminating at each domain, and those that terminate may spawn shadow and reflection rays. Shadow rays travel in the same way from the domain in which they are spawned toward each light source. We simplify reflections by choosing a uniformly random direction for each ray spawned. This simplification assumes a variety of surface orientations in each domain on which the reflection rays can be spawned.

Data I/O cost and ray communication cost are also user-defined parameters. We assume that only one data domain can be loaded at a time, so I/O costs are incurred each time a different data domain is loaded. We model asynchronous, non-blocking, point-to-point communication between processes, and we assume that each process has a separate communication thread so that work can be performed simultaneously.

6.4.2 Simulator Evaluation Methodology

Our simulator enables us to evaluate the system effects according to three parameters: cost of data I/O, cost of ray communication, and cost of ray traversal and intersection. For our tests, we chose to simulate rendering a 1024×1024 image of a data $4 \times 4 \times 4$ brick (64 data domains total) on systems of 4 processors, 16 processors and 64 processors. We ran three test sets for each system: one tracing only primary rays, one tracing primary rays plus shadow rays, and one tracing primary rays plus shadow and reflection rays.

Each test set contains four runs of each of three schedules: image decomposition, domain decomposition, and dynamic spread. The parameters for each of the four runs are listed below (parameter costs are in simulation timesteps).

baseline — I/O cost of 10^{20} per domain load; communication cost of 128 per ray sent; ray operation (traversal and intersection) costs of 1000 per ray.

large I/O — baseline, but with I/O costs increased to 10^{30}.

large comm — baseline, but with communication cost of 16384 per ray sent.

large ray ops — baseline, but with ray operation costs of 100,000 per ray.

6.4.3 Simulator Results

Since our simulator is not cycle-accurate, we present our results as the relative performance improvement (or decline) of the dynamic spread schedule versus each of the static schedules. We present our simulation results in Table 6.3. The dynamic spread schedule performed as well as or better than both static schedules for 58% (21 of 36) of our tests. For 97% (35 of 36) of our tests, the dynamic spread schedule outperformed at least one static schedule.

The dynamic spread schedule outperforms the image decomposition schedule for cases with large I/O costs and secondary rays spawned; the dynamic schedule outperforms the domain decomposition schedule for cases with high ray communication costs and no secondary rays spawned.

	4 processors							
rays	baseline		large I/O		large comm		large ray ops	
traced	Δ% img	Δ% dom	Δ% img	Δ% dom	Δ% img	Δ% dom	Δ% img	Δ% dom
primary	0	123	0	22	0	216	0	124
+ shadow	19	-247	193	6	13	68	3	82
+ reflect	13	-287	190	6	7	43	6	56

	16 processors							
rays	baseline		large I/O		large comm		large ray ops	
traced	Δ% img	Δ% dom	Δ% img	Δ% dom	Δ% img	Δ% dom	Δ% img	Δ% dom
primary	0	160	0	77	0	700	0	150
+ shadow	60	19	740	-21	27	141	5	83
+ reflect	35	-101	748	-25	-20	12	9	59

	64 processors							
rays	baseline		large I/O		large comm		large ray ops	
traced	Δ% img	Δ% dom	Δ% img	Δ% dom	Δ% img	Δ% dom	Δ% img	Δ% dom
primary	-176	63	-1	1	-405	461	-189	37
+ shadow	78	43	4959	2	-180	87	-116	47
+ reflect	38	34	4509	2	-525	-6	-173	33

Table 6.3: Relative performance improvement of the dynamic spread schedule over the image decomposition schedule (Δ% **img**) and over the domain decomposition schedule (Δ% **dom**).

The dynamic spread schedule performs significantly worse than the image decomposition schedule only when two conditions are present: there are many processors across which to parallelize the image decomposition; and data I/O costs are not the dominant factor in the total computation cost. Note the performance for the 64 processor test cases: when the I/O cost parameter is small, the dynamic spread schedule under-performs the image decomposition schedule; however, when the I/O cost parameter is large, the dynamic spread schedule becomes more competitive when only primary rays are traced and wins significantly when secondary rays are traced.

The dynamic spread schedule performs worse than the domain decomposition schedule only when three conditions are present: there are few processors, all parameters are at their small values (the baseline case) and secondary rays are spawned. Note the performance for the 4 processor test cases: the dynamic spread schedule under-performs the domain decomposition when all parameters are at their small values and when secondary rays are spawned; but when any of the parameter values are set to their large value, the dynamic spread schedule outperforms the domain decomposition.

6.5 Full Implementation Methodology

This section describes our experimental methodology, including the hardware platform, the datasets, and the rendering methods that we use to evaluate our implemented scheduling strategies.

6.5.1 Hardware Configuration

All experiments were run on *Longhorn*, a 2048 core, 256 node distributed cluster hosted at the Texas Advanced Computing Center. Each node contains two four-core Intel Xeon E5540 "Gainestown" processors and 48 GB of local RAM. All nodes are connected via a Mellanox QDR InfiniBand switch, and we use MVAPICH2 v1.4 for our MPI implementation. Our ray tracer is implemented within VisIt [21], a visualization tool designed to operate in parallel on large-scale data. We use the VisIt infrastructure to load data and to generate isosurfaces; we implemented all code related to ray tracing and ray scheduling. To focus on the effects of the schedules, we turn off all caching within the VisIt infrastructure, so that only one dataset is maintained per process. Each load of non-resident data accesses the I/O system.

All MPI communication in our implementation is two-way asynchronous. This implementation decision impacts dynamic schedules most, since they have the highest degree of communication among processes. We anticipate that moving to a one-way communication model will further increase the relative performance of dynamic schedules over static schedules.

6.5.2 Datasets

We evaluate our approach using two types of scientific data: a particle density field from an n-body cosmological simulation and a high-resolution CT scan of an abdominal cavity. For our comparative evaluation, we use a 512^3 resolution version of each dataset. For our scaling evaluation, we use a separate 6144^3 resolu-

tion cosmology dataset and a version of the 512^3 CT scan upscaled to 4096^3. The particular data sizes and decompositions on disk are presented in Table 6.2, and sample images of the data are in Figures 6.6–6.7.

For each dataset, we extract an isosurface using VisIt's VTK-based isosurfacing, then ray trace the returned geometry using two directional lights and, for the n-body particle data, two-bounce reflections. While the isosurface extraction is performed each time the dataset is loaded from disk, the cost is small relative to the I/O cost. For the n-body particle density field, we also perform direct volume ray casting, as described by Levoy [65, 66].

We study the effects of the disk decomposition of the data in Section 6.6.2. The results presented in Section 6.6 are for the decompositions that yielded fastest runtime for all tested schedules. As the results show, the decomposition that were most beneficial to the static schedules also were most beneficial to the dynamic schedules, and the dynamic schedules were able to exploit the benefit for faster runtimes relative to the static schedules.

6.6 Results

This section presents results of our ray tracer and four scheduling strategies on several scientific datasets. Our primary results are based on rendering 2048×2048 images of each dataset. In addition, we provide results that estimate the impact of a more optimized ray tracer by rendering 512×512 images of each dataset, which keeps data access costs about the same while reducing the per-ray computation cost. The difference between the primary results and the 512×512 results thus illustrate

the impact of advances in per-ray optimizations and in improved CPU performance (relative to memory bandwidth).

We evaluate our algorithm using two data sets, one consisting of particle densities from a cosmological n-body simulation, and the other an isosurface extracted from a CT scan of an abdominal cavity. We render the particle densities in two ways: (1) using direct volume ray casting and (2) performing ray tracing on an extracted isosurface.

Figure 6.8 shows results for direct volume ray casting on the n-body particle density dataset. When data load costs dominate the total computation cost, our dynamic scheduling approach outperforms static scheduling by 7× to 13× across tested process ranges. If ray computation costs dominate, the speed-up achieved by our dynamic scheduling approach is reduced to 2× to 4× better than a static scheduling strategy.

Dynamic scheduling also provides significant performance gains for ray traced isosurfaces, as shown in Figure 6.9 and Figure 6.10. When data load costs dominate the computation, ray-weighted dynamic scheduling improves runtimes 8× to 14× over static scheduling. If ray computation costs are large, ray-weighted dynamic scheduling improves runtimes 3× to 5× over static scheduling. If data load costs are relatively low, as they are with the CT scan dataset, ray-weighted dynamic scheduling runs roughly 10% faster than a static domain decomposition, which benefits from most processes having only one domain assigned, and ray-weighted dynamic scheduling still runs 2× than a static image plane decomposition. However, if ray costs dominate, the image decomposition can be as much as

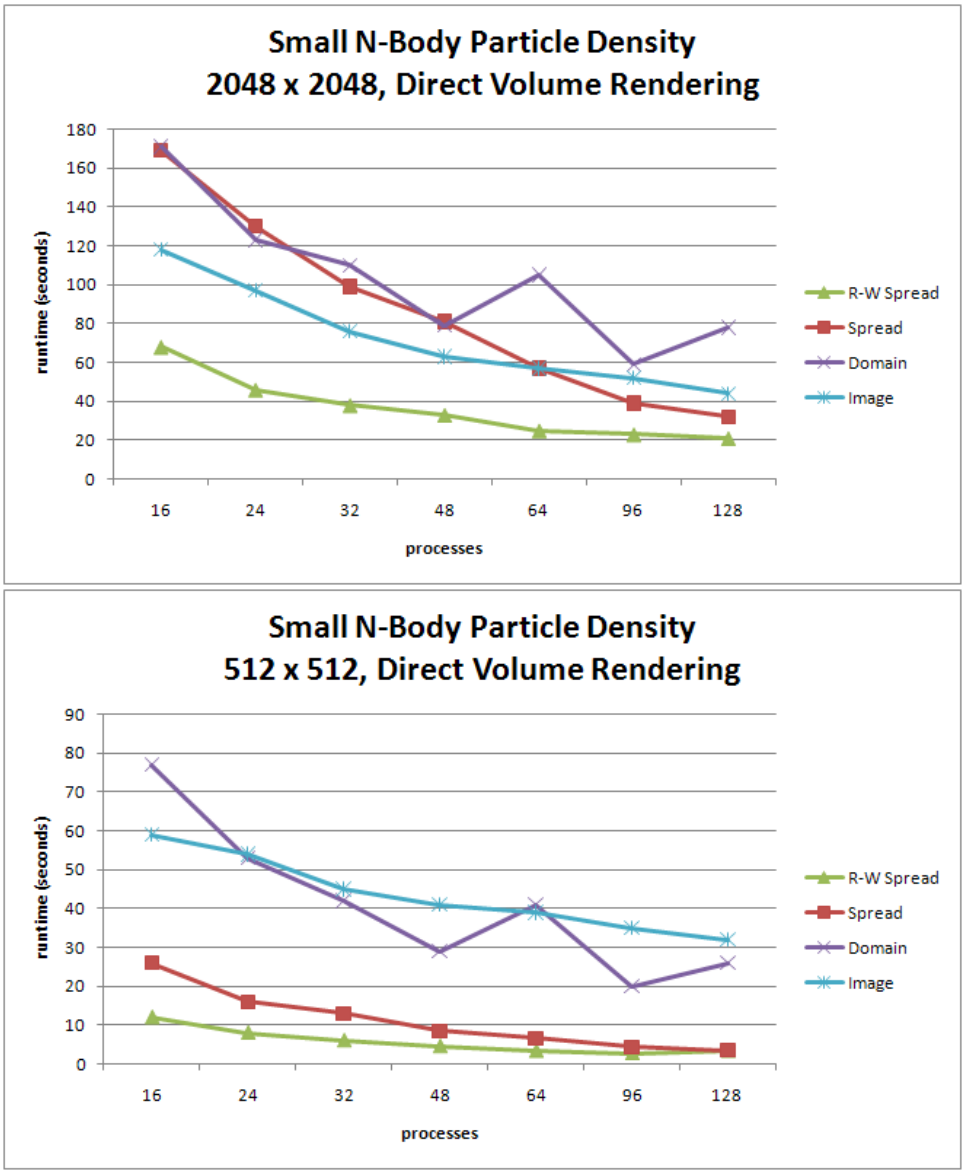

Figure 6.8: **Direct Volume Ray Casting of N-Body Particle Density** — Schedule performance for direct volume ray casting on a 512^3 particle density field. As ray calculation cost are reduced, the impact of the particular scheduling algorithm increases. When ray calculation costs are low relative to data I/O cost (512×512 case), our dynamic scheduling method can improve runtime by an order of magnitude over static schedules. Runtime is seconds per frame.

78% faster than dynamic scheduling, since the I/O costs are amortized over many ray calculations. While important to note, this case is an outlier in two respects: our tracer is unoptimized, artificially inflating the cost of ray calculations; and the abdominal CT scan isosurface is our smallest dataset, and our technique is targeted at much larger data. However, this result suggests that if the data can reside completely in memory, an image decomposition is a competitive technique.

Our performance gains are due primarily to the ability of our dynamic scheduling approach to reduce data domains loads from disk. As shown in Figure 6.11, ray-weighted dynamic scheduling can reduce data loads by $10\times$ to $48\times$, regardless of ray computation load. We note that the image decomposition schedule always touches more domains as the number of processes increases, since each process must load a domain if even one ray requires it. Under the domain decomposition schedule, a process will repeatedly swap among its assigned sub-domains. This swapping only stops when there are sufficient processes available to assign a single domain to most processes, as is the case with the Abdominal CT scan dataset.

6.6.1 Scaling

To test the scalability of our approach, we ran the ray-weighted spread schedule on large versions of our datasets (exact details are in Table 6.2). The static schedules failed to run to completion on these larger datasets. The domain decomposition schedule failed when the ray queue became too large for a particular process to contain all queued rays and the currently-loaded sub-domain. The image plane decomposition schedule failed to finish within runtime limits on *Longhorn*.

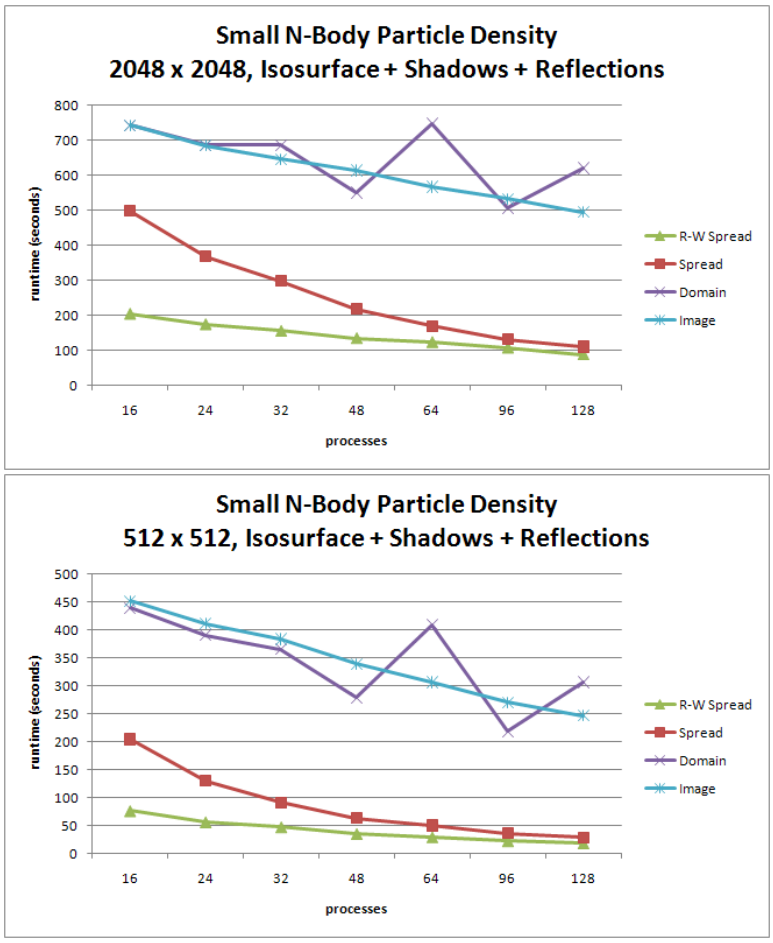

Figure 6.9: **Ray Tracing of N-Body Particle Density Isosurface** — Schedule performance for ray tracing on an isosurface of a 512^3 particle density field. The render includes two-bounce reflections and shadow rays for two directional lights. Our dynamic scheduling method can improve runtime by an order of magnitude over static schedules. Runtime is seconds per frame.

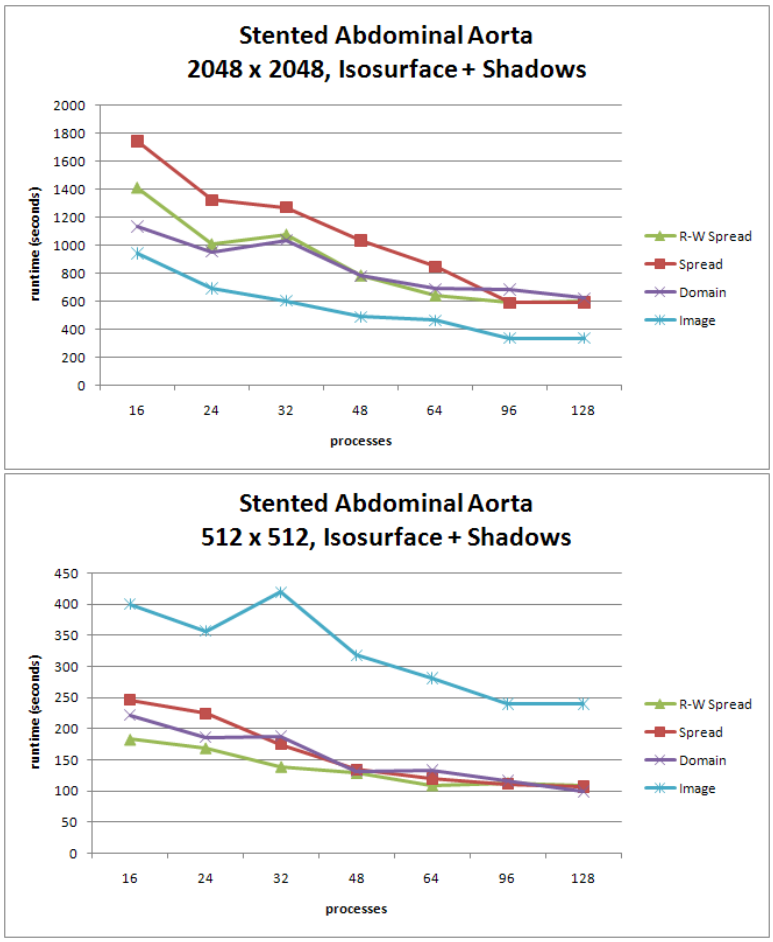

Figure 6.10: **Ray Tracing of Abdominal CT scan** — Schedule performance for ray tracing on an isosurface of a 512^3 abdominal CT scan. The render includes shadow rays for two directional lights. When data I/O costs dominate, the domain decomposition performs on-par with our dynamic schedule because data load costs are relatively small. Also, as the number of processes approaches the number of domains, more domains are loaded only once and remain resident throughout the execution. However, the image decomposition schedule still suffers from poor I/O efficiency even in this case, since domains are loaded on each process where rays require it. Yet, when ray costs dominate, the image decomposition performs best since the I/O costs can be amortized across many rays. Runtime is seconds per frame.

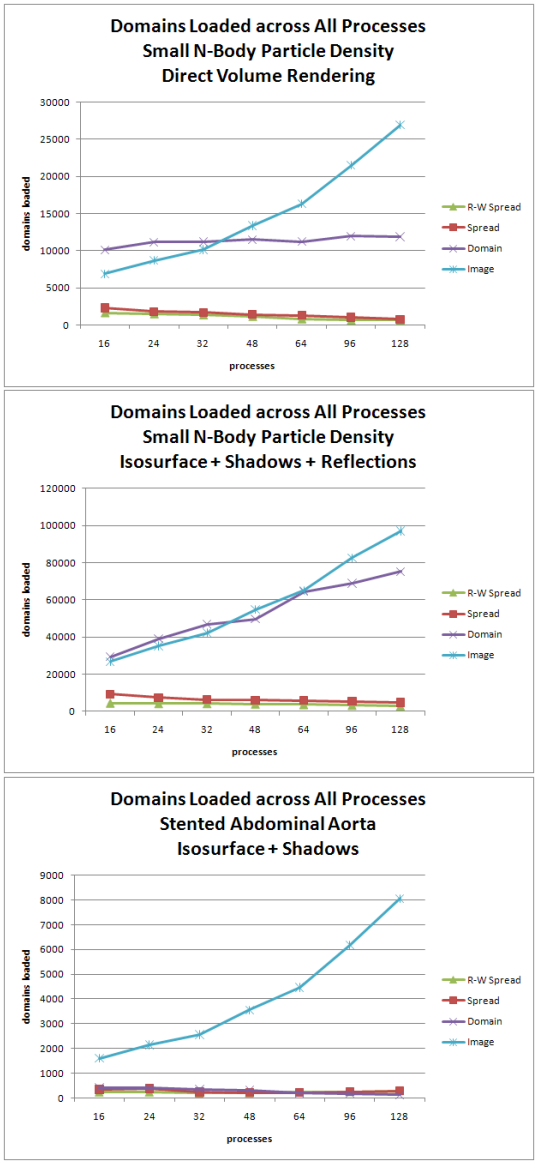

Figure 6.11: **Spatial Domains Loaded from Disk** — Number of domains (spatial subdivisions) loaded from disk for each schedule for our three test cases. Our dynamic scheduling method significantly reduces the number of domains loaded from disk, which is the primary factor for the performance gain of our approach. A similar number of domains are accessed in our two ray cost models.

Results for the ray-weighted dynamic schedule when tracing large data are presented in Figure 6.12. We believe that the increased runtime for the largest number of processes tested is due to increased communication overhead.

In Figure 6.13, we compare strong scaling speed-up for ray-weighted dynamic scheduling and for image plane decomposition. Ray-weighted dynamic scheduling demonstrates monotonically increasing speed-up until the scaling limit of the problem size is reached. The slope of the speed-up line might be improved with additional optimizations for ray operations and for ray communication among processes.

6.6.2 Effects of Decomposition on Disk

Because the I/O system has a significant impact on performance, we evaluate the effects of the decomposition of the dataset on disk by putting our two n-body datasets through two levels of subdivision. We find that dividing the disk representation of the data into spatially distinct sub-domains can improve scheduling performance. Too few sub-domains limit the flexibility of the scheduler, while too many sub-domains fragment the data and limit the number of rays queued at each domain. Empirically, a good balance seems to be struck when the number of sub-domains is several times the number of processes used to render them. The runtimes presented in Table 6.4 are for direct volume ray casting.

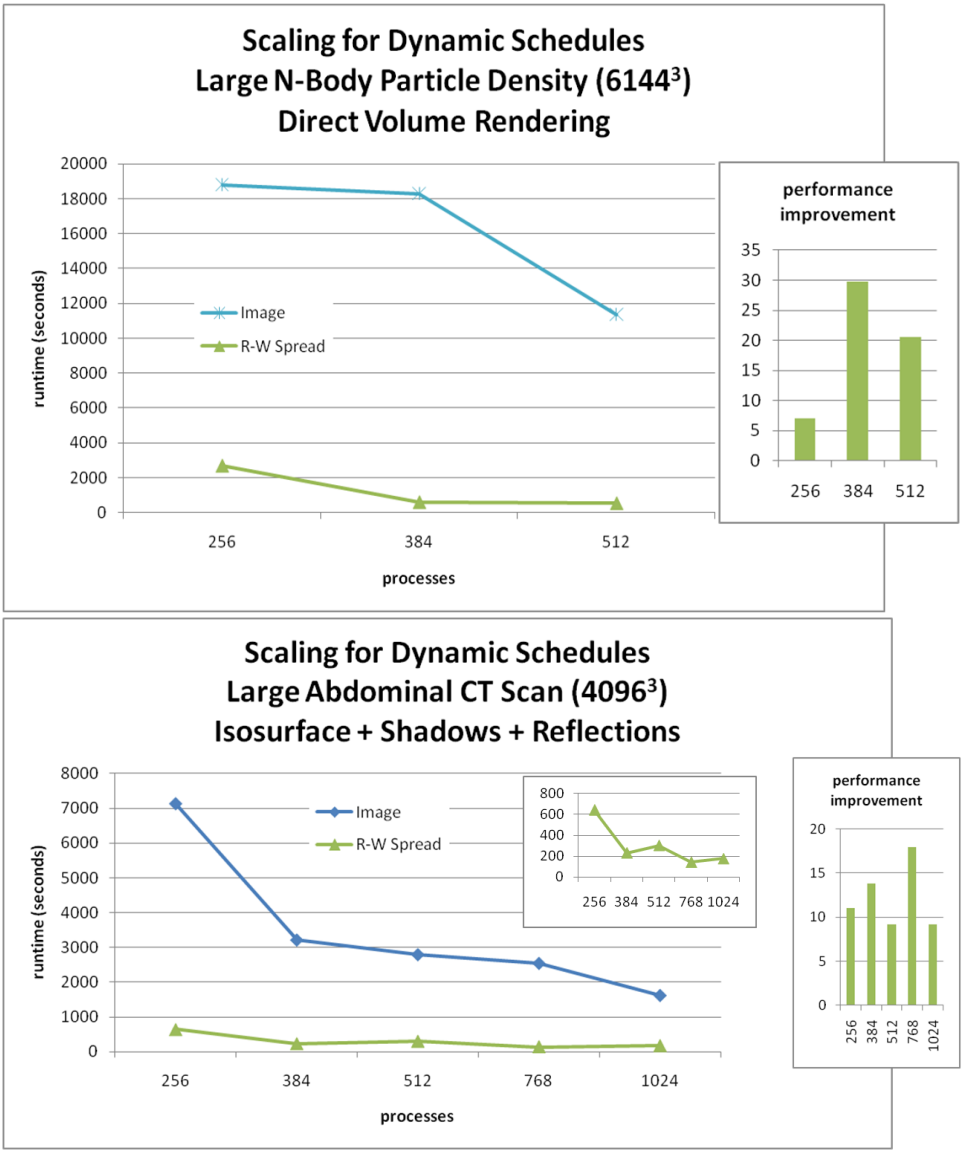

Figure 6.12: **Dynamic Schedules for Large Data** — Dynamic schedule performance for direct volume ray casting of a 6144^3 n-body particle density field and for ray tracing of an isosurface extracted from a 4096^3 abdominal CT scan. The isosurface render includes shadow rays for two directional lights and two-bounce specular reflections. Runtime is seconds per frame.

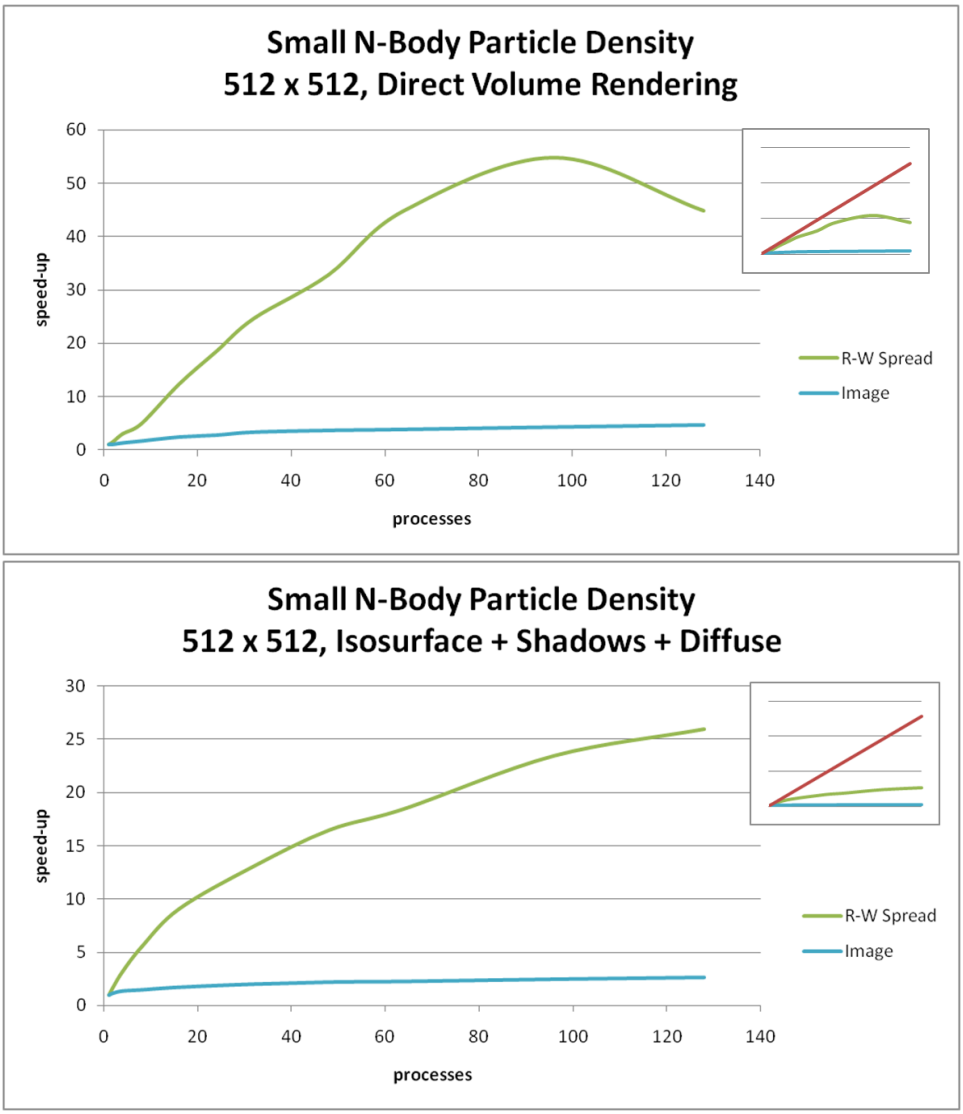

Figure 6.13: **Schedule Speed-Up** — Image decomposition and ray-weighted spread schedule performance for direct volume ray casting and ray tracing an isosurface of a 512^3 n-body particle density field. The isosurface render includes shadow rays for two directional lights and $16\times$ sampled, two-bounce diffuse reflections.

512^3 N-Body Particle Density
128 processes

sub-domains	n=8	64	512
MB per sub-domain	1100	130	17
R-W Spread	151	54	21
Spread	156	54	32
Domain	186	86	78
Image	92	56	44

6144^3 N-Body Particle Density
256 processes

sub-domains	n=512	4096	32768
MB per sub-domain	3400	436	55
R-W Spread	2881	2164	6340
Spread	—	—	—
Domain	—	—	—
Image	—	—	—

Table 6.4: **Effects of Disk Decomposition on Runtimes** — Effect of the number of data files on scheduling performance. The runtimes presented below are for direct volume ray casting. Cases marked '—' failed to execute successfully, due to exceeding either memory limits or runtime limits.

6.7 Summary

In this chapter, we have presented a dynamic scheduling approach to large-scale distributed memory ray tracing. Our ray-weighted dynamic schedule is robust across many data sizes and rendering modes, and our approach can achieve an order of magnitude speed-up over static scheduling methods when data access costs dominate the computation. In addition, our dynamic schedule can render datasets that cannot be rendered by a static schedule. Our ray tracer has not been thoroughly optimized, and we have argued that as ray calculation costs are reduced through optimization, the relative performance of our approach will improve further against competing techniques.

We have just begun to explore the space of possible dynamic schedules, and further work is warranted to identify schedules that achieve particular system goals. In particular, it may be possible to schedule a sub-domain across several available processes, though this may increase both I/O and communication costs. A dynamic schedule could also speculatively load data based on anticipated ray travel, particularly for an animation sequence where rendering information from the previous frame is available.

Chapter 7

Conclusion

In this dissertation, we have presented *dynamic ray scheduling*, a new formulation of ray tracing that treats both rays and scene data as first-class schedulable elements. This interpretation creates a spectrum of possible scheduling strategies to optimize data movement and computation. In addition, it incorporates previous ray tracing algorithms as special-case schedules. We demonstrate that dynamically scheduled ray tracing provides the flexibility to optimize rendering performance across a wide range of hardware configurations.

First, in Chapter 3, we presented an analytic model of the bandwidth consumed by recursive ray tracing, which explicitly defines the factors that cause excess bandwidth consumption. We demonstrated that bandwidth consumption is driven by the rate at which data is replaced in memory, which in turn is primarily determined by the amount of scene data accessed when traversing the acceleration structure. A traversal step to a "large leaf", a node that evicts a significant portion of resident data from memory, can trigger a chain-reaction of evictions if many rays have that traversal step on their path. The model predicts that if several paths contain large leaves, their traversal can cause thrashing and excessive bandwidth consumption. We validated our model against an efficient but unoptimized research

ray tracer, and we discovered that small memory sizes amplified the effect of large leaves, potentially increasing bandwidth consumed by $10\times$ to $100\times$.

Our analytic model suggested a bandwidth bottleneck for recursive ray tracing, and in Chapter 4, we confirmed the existence of this bottleneck. Using an instrumented, fully-optimized recursive ray tracer [91], we showed that recursive ray tracers are bandwidth-limited particularly when rendering large models with shading that requires tracing incoherent secondary rays. Further, this bandwidth limitation is caused by poor cache utilization rather than by a large increase in the working set. A tracer that can rearrange ray computations in order to build additional ray coherence might achieve better cache utilization, and thereby achieve better system performance. We noted that queueing tracers [85] can provide this algorithmic flexibility.

In Chapter 5, we presented our dynamic scheduling algorithm for a single core processor, which targets memory traffic between DRAM and the lowest processor cache. We demonstrated that this algorithm can significantly reduce geometry traffic by as much as $100\times$, but at the cost of increased ray traffic. When tracing incoherent rays with scarce memory resources, our algorithm can provide a net bandwidth savings of $10\times$ or more. When memory is plentiful, our algorithm consumes more bandwidth than a recursive tracer, but keeps total bandwidth consumption low.

We showed in Chapter 6 that our dynamic scheduling algorithm can efficiently render large-scale data on distributed-memory clusters by targeting disk to DRAM bandwidth. Our dynamic approach can render large-scale data over $10\times$

faster than statically scheduled approaches, and our approach demonstrates better scaling. As in the single-core case, the distributed-memory version of our dynamic algorithm can consume more bandwidth than a static algorithm for cases where memory resources are plentiful, but the absolute bandwidth consumed by our algorithm in these cases remains small.

Through this dissertation, we have demonstrated that our approach can significantly improve memory system efficiency over recursive algorithms, both for the cache of a single processor and across the aggregate distributed memory of a large-scale cluster, while reducing the overhead present in other queueing tracers. Our new interpretation of ray tracing algorithms as schedulers opens new opportunities for algorithm development. In particular, we can explore additional optimizations for data locality from compiler literature [7, 20, 69, 73, 120] and additional scheduling ideas from the thread scheduling literature [11, 86].

We would like to integrate our dynamic scheduling algorithm into general ray tracing frameworks [16, 81, 84] to leverage the considerable effort that has gone into them, but because these frameworks are built around the recursive algorithm, it is not clear how to do so without breaking many optimizations intrinsic to their performance. Our algorithm can be applied to ray tracing on other memory-constrained parallel systems, such as on rasterizing graphics hardware [3, 4, 19, 81, 88]. Aila and Karras provide excellent initial work in this direction by implementing dynamic scheduling for a hypothetical GPU-like architecture [3]. Also, Budge et al. present a queueing path tracer that renders large scenes using the combined resources of distributed CPUs and GPUs across a small cluster [19].

Looking to the future, we also would like to expand dynamic scheduling to animation sequences, where information from previous frames can inform the scheduler and provide additional performance gain. Since the data to be rendered tends to remain constant between animation frames [102], information gathered when rendering previous frames can be powerful guides for the scheduler to distribute work effectively.

Appendices

Appendix A

No-Cache Bandwidth Equation

The amount of bandwidth consumed without caching is the upper bound on bandwidth consumption for a system. We will use the general bandwidth equation (Equation 3.1) to determine, first, the bandwidth consumption per ray, and, second, the bandwidth consumed by the rendering process. This equation will form the basis for our equations in Section 3.4.4 that consider caching effects. Please refer to Tables 3.1 and 3.2 for the definition of equation parameters and derived terms.

Equation 3.1 shows that bandwidth consumption is the sum of the ray contribution (BW_r), the acceleration structure node contribution (BW_n), and the geometry contribution (BW_g). For a single ray, the ray contribution is simply the cost of loading that single ray, represented as C_r. Thus, $BW_r = C_r$. The ray must be tested against some number n of acceleration structure nodes during traversal. If the cost of loading each node is C_n, then the node contribution $BW_n = nC_n$. If the ray reaches any leaf of the acceleration structure, the ray must be tested for intersection against the geometry at that leaf. We assume that each leaf of the acceleration structure has g geometric objects (recall that g is the expected amount of geometry at each node). We use C_g to represent the cost to load each geometric object. So if the ray reaches l leaves (where $l \leq n$), each of which has g geometric objects, the geometry

contribution is $BW_g = lgC_g$. We combine these three contributions in Equation A.1, which describes the bandwidth cost per ray absent any caching.

$$BW_{per\ ray} = C_r + nC_n + lgC_g \tag{A.1}$$

Since this case assumes no caching, all node and geometry data must be loaded for each ray. Therefore, we multiply the per-ray equation (Equation A.1) by the number of rays traced R to determine the bandwidth consumed by the rendering process (Equation A.2). Note that it is unlikely that all rays will traverse exactly the same number of nodes or reach the same number of leaf nodes. We use expected values for the node count n and leaf count l, which estimate normal behavior. It is possible to choose a camera position and direction for which the actual nodes and leaves touched will vary significantly from the expected values.

$$BW = R\left[C_r + nC_n + lgC_g\right] \tag{A.2}$$

Appendix B

Tabular Results for Single Core Ray Tracer

room — 47K triangles (5.76 MB)
primary rays only — 6.7K triangles (0.82 MB) potentially visible

avail cache	frac of pot-vis	geometry traffic (MB) recursive	packet	dynamic	total traffic (MB) recursive	packet	dynamic
1 K	0.001	1190.51	661.19	612.05	1254.51	725.19	889.37
2 K	0.002	705.89	331.50	297.29	769.89	395.50	652.42
4 K	0.005	327.04	186.61	170.71	391.04	250.61	516.08
8 K	0.010	142.32	90.95	83.64	206.32	154.95	430.08
16 K	0.019	2.76	4.30	1.28	66.76	68.30	318.91
32 K	0.038	1.13	3.24	1.11	65.13	67.24	297.08
64 K	0.076	1.02	1.65	0.97	65.02	65.65	264.60
128 K	0.153	0.93	0.83	0.89	64.93	64.83	188.59
256 K	0.306	0.90	0.82	0.86	64.90	64.82	141.26
512 K	0.612	0.84	0.82	0.83	64.84	64.82	105.32
1024 K	1.224	0.82	0.82	0.82	64.82	64.82	102.15
2048 K	2.448	0.82	0.82	0.82	64.82	64.82	65.16
4096 K	4.896	0.82	0.82	0.82	64.82	64.82	64.82
avail cache	frac of pot-vis	relative improvement (geometry) recursive	packet	dynamic	relative improvement (total) recursive	packet	dynamic
1 K	0.001	1.00	1.80	**1.95**	1.00	1.73	**1.41**
2 K	0.002	1.00	2.13	**2.37**	1.00	1.95	**1.18**
4 K	0.005	1.00	1.75	**1.92**	1.00	1.56	0.76
8 K	0.010	1.00	1.56	**1.70**	1.00	1.33	0.48
16 K	0.019	1.00	0.64	**2.16**	1.00	0.98	0.21
32 K	0.038	1.00	0.35	**1.02**	1.00	0.97	0.22
64 K	0.076	1.00	0.62	**1.05**	1.00	0.99	0.25
128 K	0.153	1.00	1.12	1.04	1.00	1.00	0.34
256 K	0.306	1.00	1.10	1.05	1.00	1.00	0.46
512 K	0.612	1.00	1.02	1.01	1.00	1.00	0.62
1024 K	1.224	1.00	1.00	1.00	1.00	1.00	0.63
2048 K	2.448	1.00	1.00	1.00	1.00	1.00	0.99
4096 K	4.896	1.00	1.00	1.00	1.00	1.00	1.00

Table B.1: Traffic from main memory to processor cache for **room** for primary rays. Larger numbers indicate larger factors of improvement.

room — 47K triangles (5.76 MB)
primary + hard shadow rays — 10.0K triangles (1.22 MB) potentially visible

avail cache	frac of pot-vis	geometry traffic (MB) recursive	packet	dynamic	total traffic (MB) recursive	packet	dynamic
1 K	0.001	6503.33	5952.17	2235.40	6567.33	6016.17	2668.93
2 K	0.002	4902.58	3798.42	1143.81	4966.58	3862.42	1655.15
4 K	0.003	2854.40	1642.33	628.22	2918.40	1706.33	1129.82
8 K	0.006	1050.32	680.11	289.06	1114.32	744.11	791.77
16 K	0.013	45.60	23.45	4.65	109.60	87.45	478.53
32 K	0.026	4.63	11.17	4.05	68.63	75.17	456.45
64 K	0.051	3.31	8.65	3.51	67.31	72.65	424.16
128 K	0.102	2.78	4.39	3.15	66.78	68.39	347.25
256 K	0.205	2.46	1.94	2.99	66.46	65.94	301.12
512 K	0.409	2.05	1.70	2.41	66.05	65.70	269.90
1024 K	0.819	1.32	1.35	0.88	65.32	65.35	271.00
2048 K	1.638	1.22	1.22	0.88	65.22	65.22	221.76
4096 K	3.276	1.22	1.22	0.88	65.22	65.22	221.08
avail cache	frac of pot-vis	relative improvement (geometry) recursive	packet	dynamic	relative improvement (total) recursive	packet	dynamic
1 K	0.001	1.00	1.09	**2.91**	1.00	1.09	**2.46**
2 K	0.002	1.00	1.29	**4.29**	1.00	1.29	**3.00**
4 K	0.003	1.00	1.74	**4.54**	1.00	1.71	**2.58**
8 K	0.006	1.00	1.54	**3.63**	1.00	1.50	**1.41**
16 K	0.013	1.00	1.94	**9.81**	1.00	1.25	0.23
32 K	0.026	1.00	0.41	**1.14**	1.00	0.91	0.15
64 K	0.051	1.00	0.38	0.94	1.00	0.93	0.16
128 K	0.102	1.00	0.63	0.88	1.00	0.98	0.19
256 K	0.205	1.00	1.27	0.82	1.00	1.01	0.22
512 K	0.409	1.00	1.21	0.85	1.00	1.01	0.24
1024 K	0.819	1.00	0.98	**1.50**	1.00	1.00	0.24
2048 K	1.638	1.00	1.00	**1.39**	1.00	1.00	0.29
4096 K	3.276	1.00	1.00	**1.39**	1.00	1.00	0.30

Table B.2: Traffic from main memory to processor cache for **room** for primary + hard shadow rays. Larger numbers indicate larger factors of improvement.

room — 47K triangles (5.76 MB)
primary + hard shadow + specular reflection rays
14.4K triangles (1.76 MB) potentially visible

avail cache	frac of pot-vis	geometry traffic (MB) recursive	packet	dynamic	total traffic (MB) recursive	packet	dynamic
1 K	0.001	24319.32	22343.64	9520.93	24383.32	22407.64	9990.26
2 K	0.001	19825.02	15450.62	5581.49	19889.02	15514.62	6128.63
4 K	0.002	15699.61	8993.06	3786.84	15763.61	9057.06	4324.23
8 K	0.004	9894.89	4758.64	1750.56	9958.89	4822.64	2289.07
16 K	0.009	3011.72	862.27	30.28	3075.72	926.27	539.95
32 K	0.018	326.39	355.33	25.55	390.39	419.33	513.74
64 K	0.036	168.02	193.90	20.90	232.02	257.90	477.34
128 K	0.071	103.16	133.41	17.68	167.16	197.41	399.42
256 K	0.142	62.06	95.12	15.70	126.06	159.12	356.66
512 K	0.284	30.10	45.64	11.53	94.10	109.64	334.52
1024 K	0.569	7.42	8.15	4.93	71.42	72.15	330.56
2048 K	1.138	1.76	1.76	1.63	65.76	65.76	258.47
4096 K	2.276	1.76	1.76	1.63	65.76	65.76	257.68

avail cache	frac of pot-vis	relative improvement (geometry) recursive	packet	dynamic	relative improvement (total) recursive	packet	dynamic
1 K	0.001	1.00	1.09	**2.55**	1.00	1.09	**2.44**
2 K	0.001	1.00	1.28	**3.55**	1.00	1.28	**3.25**
4 K	0.002	1.00	1.75	**4.15**	1.00	1.74	**3.65**
8 K	0.004	1.00	2.08	**5.65**	1.00	2.07	**4.35**
16 K	0.009	1.00	3.49	**99.01**	1.00	3.32	**5.70**
32 K	0.018	1.00	0.92	**12.77**	1.00	0.93	0.76
64 K	0.036	1.00	0.87	**8.04**	1.00	0.90	0.49
128 K	0.071	1.00	0.77	**5.83**	1.00	0.85	0.42
256 K	0.142	1.00	0.65	**3.95**	1.00	0.79	0.35
512 K	0.284	1.00	0.66	**2.61**	1.00	0.86	0.28
1024 K	0.569	1.00	0.91	**1.51**	1.00	0.99	0.22
2048 K	1.138	1.00	1.00	**1.08**	1.00	1.00	0.25
4096 K	2.276	1.00	1.00	**1.08**	1.00	1.00	0.26

Table B.3: Traffic from main memory to processor cache for **room** for primary + hard shadow + specular reflection rays. Larger numbers indicate larger factors of improvement.

room — 47K triangles (5.76 MB)
primary + hard shadow + diffuse reflection rays
15.4K triangles (1.84 MB) potentially visible

avail cache	frac of pot-vis	geometry traffic (MB) recursive	packet	dynamic	total traffic (MB) recursive	packet	dynamic
1 K	0.001	27132.28	26859.52	8342.00	27148.28	26875.52	8651.32
2 K	0.001	22145.86	21716.53	3832.36	22161.86	21732.53	4161.13
4 K	0.002	17562.88	16970.06	2352.53	17578.88	16986.06	2678.87
8 K	0.004	12730.80	12230.42	1015.58	12746.80	12246.42	1342.18
16 K	0.008	8846.82	8485.41	34.85	8862.82	8501.41	354.26
32 K	0.017	6178.04	5964.22	28.90	6194.04	5980.22	342.95
64 K	0.033	3746.91	3707.78	24.19	3762.91	3723.78	331.01
128 K	0.067	1976.33	1965.16	19.90	1992.33	1981.16	307.49
256 K	0.133	821.24	819.68	17.10	837.24	835.68	294.37
512 K	0.266	225.58	230.60	12.41	241.58	246.60	285.18
1024 K	0.533	20.97	30.31	5.89	36.97	46.31	279.32
2048 K	1.065	1.84	1.88	1.76	17.84	17.88	257.98
4096 K	2.130	1.84	1.88	1.76	17.84	17.88	258.07

avail cache	frac of pot-vis	relative improvement (geometry) recursive	packet	dynamic	relative improvement (total) recursive	packet	dynamic
1 K	0.001	1.00	1.01	**3.25**	1.00	0.99	**3.14**
2 K	0.001	1.00	1.02	**5.78**	1.00	0.98	**5.33**
4 K	0.002	1.00	1.03	**7.47**	1.00	0.97	**6.56**
8 K	0.004	1.00	1.04	**12.54**	1.00	0.96	**9.50**
16 K	0.008	1.00	1.04	**253.85**	1.00	0.96	**25.02**
32 K	0.017	1.00	1.04	**213.77**	1.00	0.97	**18.06**
64 K	0.033	1.00	1.01	**154.89**	1.00	0.99	**11.37**
128 K	0.067	1.00	1.01	**99.31**	1.00	0.99	**6.48**
256 K	0.133	1.00	1.00	**48.03**	1.00	1.00	**2.48**
512 K	0.266	1.00	0.98	**18.18**	1.00	1.02	0.85
1024 K	0.533	1.00	0.69	**3.56**	1.00	1.25	0.13
2048 K	1.065	1.00	0.98	**1.05**	1.00	1.00	0.07
4096 K	2.130	1.00	0.98	**1.05**	1.00	1.00	0.07

Table B.4: Traffic from main memory to processor cache for **room** for primary + hard shadow + diffuse reflection rays. Larger numbers indicate larger factors of improvement.

grove — 164K triangles (20.11 MB)
primary rays only — 127K triangles (15.49 MB) potentially visible

avail cache	frac of pot-vis	geometry traffic (MB) recursive	packet	dynamic	total traffic (MB) recursive	packet	dynamic
1 K	< 0.001	4878.67	4759.77	4611.01	4942.67	4823.77	5198.92
2 K	< 0.001	3940.24	3274.72	2730.68	4004.24	3338.72	3310.41
4 K	< 0.001	3091.94	2006.40	1478.22	3155.94	2070.40	2011.84
8 K	0.001	2112.50	874.21	512.41	2176.50	938.21	993.60
16 K	0.001	839.94	194.23	58.30	903.94	258.23	455.63
32 K	0.002	75.01	53.69	30.61	139.01	117.69	378.94
64 K	0.004	25.63	42.82	26.71	89.63	106.82	333.69
128 K	0.008	21.60	42.18	23.66	85.60	106.18	300.77
256 K	0.016	19.21	40.68	20.53	83.21	104.68	252.69
512 K	0.032	17.78	35.07	19.08	81.78	99.07	211.12
1024 K	0.065	16.81	15.53	18.27	80.81	79.53	197.00
2048 K	0.129	16.26	15.49	17.39	80.26	79.49	185.47
4096 K	0.258	15.89	15.49	16.38	79.89	79.49	175.73
avail cache	frac of pot-vis	relative improvement (geometry) recursive	packet	dynamic	relative improvement (total) recursive	packet	dynamic
1 K	< 0.001	1.00	1.02	**1.06**	1.00	1.02	0.95
2 K	< 0.001	1.00	1.20	**1.44**	1.00	1.20	1.21
4 K	< 0.001	1.00	1.54	**2.09**	1.00	1.52	1.57
8 K	0.001	1.00	2.42	**4.12**	1.00	2.32	2.19
16 K	0.001	1.00	4.32	**14.41**	1.00	3.50	1.98
32 K	0.002	1.00	1.40	**2.45**	1.00	1.18	0.37
64 K	0.004	1.00	0.60	0.96	1.00	0.84	0.27
128 K	0.008	1.00	0.51	0.91	1.00	0.81	0.28
256 K	0.016	1.00	0.47	0.94	1.00	0.79	0.33
512 K	0.032	1.00	0.51	0.93	1.00	0.83	0.39
1024 K	0.065	1.00	1.08	0.92	1.00	1.02	0.41
2048 K	0.129	1.00	1.05	0.94	1.00	1.01	0.43
4096 K	0.258	1.00	1.03	0.97	1.00	1.01	0.45

Table B.5: Traffic from main memory to processor cache for **grove** for primary rays. Larger numbers indicate larger factors of improvement.

grove — 164K triangles (20.11 MB)
primary + hard shadow rays — 146K triangles (17.86 MB) potentially visible

avail cache	frac of pot-vis	geometry traffic (MB) recursive	packet	dynamic	total traffic (MB) recursive	packet	dynamic
1 K	< 0.001	16456.48	16369.24	11761.99	16520.48	16433.24	12377.85
2 K	< 0.001	13453.69	12751.53	6939.68	13517.69	12815.53	7547.35
4 K	< 0.001	10932.37	9541.58	3831.61	10996.37	9605.58	4393.17
8 K	< 0.001	8349.56	6299.49	1335.52	8413.56	6363.49	1888.32
16 K	0.001	5205.51	3135.72	185.01	5269.51	3199.72	653.95
32 K	0.002	1747.25	1005.87	94.37	1811.25	1069.87	514.31
64 K	0.003	312.82	331.45	80.48	376.82	395.45	459.08
128 K	0.007	147.59	200.51	70.20	211.59	264.51	418.93
256 K	0.014	93.47	179.10	63.12	157.47	243.10	399.26
512 K	0.028	64.60	169.80	58.19	128.60	233.80	354.22
1024 K	0.056	46.52	159.46	55.15	110.52	223.46	337.96
2048 K	0.112	33.44	132.42	51.85	97.44	196.42	324.04
4096 K	0.224	26.15	38.64	48.00	90.15	102.64	311.68

avail cache	frac of pot-vis	relative improvement (geometry) recursive	packet	dynamic	relative improvement (total) recursive	packet	dynamic
1 K	< 0.001	1.00	1.01	**1.40**	1.00	1.01	**1.33**
2 K	< 0.001	1.00	1.06	**1.94**	1.00	1.05	**1.79**
4 K	< 0.001	1.00	1.15	**2.85**	1.00	1.14	**2.50**
8 K	< 0.001	1.00	1.33	**6.25**	1.00	1.32	**4.46**
16 K	0.001	1.00	1.66	**28.14**	1.00	1.65	**8.06**
32 K	0.002	1.00	1.74	**18.51**	1.00	1.69	**3.52**
64 K	0.003	1.00	0.94	**3.89**	1.00	0.95	0.82
128 K	0.007	1.00	0.74	**2.10**	1.00	0.80	0.51
256 K	0.014	1.00	0.52	**1.48**	1.00	0.65	0.39
512 K	0.028	1.00	0.38	**1.11**	1.00	0.55	0.36
1024 K	0.056	1.00	0.29	0.84	1.00	0.49	0.33
2048 K	0.112	1.00	0.25	0.64	1.00	0.50	0.30
4096 K	0.224	1.00	0.68	0.54	1.00	0.88	0.29

Table B.6: Traffic from main memory to processor cache for **grove** for primary + hard shadow rays. Larger numbers indicate larger factors of improvement.

grove — 164K triangles (20.11 MB)
primary + hard shadow + specular reflection rays
161K triangles (19.65 MB) potentially visible

avail cache	frac of pot-vis	geometry traffic (MB) recursive	packet	dynamic	total traffic (MB) recursive	packet	dynamic
1 K	< 0.001	32234.51	32097.76	24307.08	32298.51	32161.76	25086.99
2 K	< 0.001	26554.50	25509.81	15047.44	26618.50	25573.81	15819.16
4 K	< 0.001	21971.56	19602.54	8632.18	22035.56	19666.54	9357.80
8 K	< 0.001	17897.54	13787.43	3357.71	17961.54	13851.43	4030.91
16 K	0.001	13488.00	8555.37	717.55	13552.00	8619.37	1306.88
32 K	0.002	8413.83	5266.98	456.06	8477.83	5330.98	996.39
64 K	0.003	4141.86	3720.42	378.81	4205.86	3784.42	877.79
128 K	0.006	2658.29	2744.06	318.88	2722.29	2808.06	787.99
256 K	0.013	2053.98	2146.78	279.52	2117.98	2210.78	703.68
512 K	0.025	1587.80	1676.14	248.25	1651.80	1740.14	632.31
1024 K	0.051	1149.28	1302.49	224.13	1213.28	1366.49	594.96
2048 K	0.102	715.54	943.60	202.18	779.54	1007.60	562.51
4096 K	0.204	332.24	644.69	179.82	396.24	708.69	531.92

avail cache	frac of pot-vis	relative improvement (geometry) recursive	packet	dynamic	relative improvement (total) recursive	packet	dynamic
1 K	< 0.001	1.00	1.00	**1.33**	1.00	1.00	**1.29**
2 K	< 0.001	1.00	1.04	**1.76**	1.00	1.04	**1.68**
4 K	< 0.001	1.00	1.12	**2.55**	1.00	1.12	**2.35**
8 K	< 0.001	1.00	1.30	**5.33**	1.00	1.30	**4.46**
16 K	0.001	1.00	1.58	**18.80**	1.00	1.57	**10.37**
32 K	0.002	1.00	1.60	**18.45**	1.00	1.59	**8.51**
64 K	0.003	1.00	1.11	**10.93**	1.00	1.11	**4.79**
128 K	0.006	1.00	0.97	**8.34**	1.00	0.97	**3.45**
256 K	0.013	1.00	0.96	**7.35**	1.00	0.96	**3.01**
512 K	0.025	1.00	0.95	**6.40**	1.00	0.95	**2.61**
1024 K	0.051	1.00	0.88	**5.13**	1.00	0.89	**2.04**
2048 K	0.102	1.00	0.76	**3.54**	1.00	0.77	**1.39**
4096 K	0.204	1.00	0.52	**1.85**	1.00	0.56	0.74

Table B.7: Traffic from main memory to processor cache for **grove** for primary + hard shadow + specular reflection rays. Larger numbers indicate larger factors of improvement.

grove — 164K triangles (20.11 MB)
primary + hard shadow + diffuse reflection rays
161K triangles (19.62 MB) potentially visible

avail cache	frac of pot-vis	geometry traffic (MB) recursive	packet	dynamic	total traffic (MB) recursive	packet	dynamic
1 K	< 0.001	23581.31	23567.99	19718.37	23597.31	23583.99	20105.27
2 K	< 0.001	19448.90	19350.91	12368.50	19464.90	19366.91	12753.37
4 K	< 0.001	16195.96	15946.00	7247.89	16211.96	15962.00	7621.18
8 K	< 0.001	13492.55	13017.76	3037.43	13508.55	13033.76	3397.65
16 K	0.001	11276.60	10581.27	801.17	11292.60	10597.27	1140.42
32 K	0.002	9654.98	8963.49	539.21	9670.98	8979.49	866.23
64 K	0.003	8359.51	7950.33	440.57	8375.51	7966.33	757.26
128 K	0.006	7013.85	6892.24	360.19	7029.85	6908.24	669.42
256 K	0.013	5741.27	5687.78	315.06	5757.27	5703.78	613.09
512 K	0.025	4435.86	4393.57	276.74	4451.86	4409.57	564.75
1024 K	0.051	3008.07	3055.72	245.21	3024.07	3071.72	529.93
2048 K	0.102	1631.21	1682.71	218.59	1647.21	1698.71	500.76
4096 K	0.204	615.06	748.40	182.56	631.06	764.40	463.14

avail cache	frac of pot-vis	relative improvement (geometry) recursive	packet	dynamic	relative improvement (total) recursive	packet	dynamic
1 K	< 0.001	1.00	1.00	**1.20**	1.00	1.00	**1.17**
2 K	< 0.001	1.00	1.01	**1.57**	1.00	0.99	**1.53**
4 K	< 0.001	1.00	1.02	**2.23**	1.00	0.98	**2.13**
8 K	< 0.001	1.00	1.04	**4.44**	1.00	0.96	**3.98**
16 K	0.001	1.00	1.07	**14.08**	1.00	0.94	**9.90**
32 K	0.002	1.00	1.08	**17.91**	1.00	0.93	**11.16**
64 K	0.003	1.00	1.05	**18.97**	1.00	0.95	**11.06**
128 K	0.006	1.00	1.02	**19.47**	1.00	0.98	**10.50**
256 K	0.013	1.00	1.01	**18.22**	1.00	0.99	**9.39**
512 K	0.025	1.00	1.01	**16.03**	1.00	0.99	**7.88**
1024 K	0.051	1.00	0.98	**12.27**	1.00	1.02	**5.71**
2048 K	0.102	1.00	0.97	**7.46**	1.00	1.03	**3.29**
4096 K	0.204	1.00	0.82	**3.37**	1.00	1.21	**1.36**

Table B.8: Traffic from main memory to processor cache for **grove** for primary + hard shadow + diffuse reflection rays. Larger numbers indicate larger factors of improvement.

sphereflake — 797K triangles (97.31 MB)
primary rays only — 47K triangles (5.74 MB) potentially visible

avail cache	frac of pot-vis	geometry traffic (MB) recursive	packet	dynamic	total traffic (MB) recursive	packet	dynamic
1 K	< 0.001	1318.02	1070.85	1019.79	1382.02	1134.85	1263.22
2 K	< 0.001	999.68	664.19	596.86	1063.68	728.19	817.53
4 K	0.001	506.36	212.43	157.68	570.36	276.43	372.15
8 K	0.001	152.01	70.89	48.55	216.01	134.89	247.83
16 K	0.003	17.34	17.31	7.77	81.34	81.31	234.10
32 K	0.005	8.16	14.63	7.13	72.16	78.63	217.31
64 K	0.011	7.08	13.87	6.74	71.08	77.87	234.71
128 K	0.022	6.53	10.03	6.31	70.53	74.03	221.64
256 K	0.044	6.17	6.66	6.07	70.17	70.66	189.51
512 K	0.087	5.90	5.74	5.88	69.90	69.74	169.82
1024 K	0.174	5.80	5.74	5.80	69.80	69.74	136.77
2048 K	0.348	5.78	5.74	5.80	69.78	69.74	127.40
4096 K	0.697	5.75	5.74	5.75	69.75	69.74	107.07
avail cache	frac of pot-vis	relative improvement (geometry) recursive	packet	dynamic	relative improvement (total) recursive	packet	dynamic
1 K	< 0.001	1.00	1.23	**1.29**	1.00	1.22	1.09
2 K	< 0.001	1.00	1.51	**1.67**	1.00	1.46	1.30
4 K	0.001	1.00	2.38	**3.21**	1.00	2.06	1.53
8 K	0.001	1.00	2.14	**3.13**	1.00	1.60	0.87
16 K	0.003	1.00	1.00	**2.23**	1.00	1.00	0.35
32 K	0.005	1.00	0.56	**1.14**	1.00	0.92	0.33
64 K	0.011	1.00	0.51	**1.05**	1.00	0.91	0.30
128 K	0.022	1.00	0.65	**1.03**	1.00	0.95	0.32
256 K	0.044	1.00	0.93	**1.02**	1.00	0.99	0.37
512 K	0.087	1.00	1.03	1.00	1.00	1.00	0.41
1024 K	0.174	1.00	1.01	1.00	1.00	1.00	0.51
2048 K	0.348	1.00	1.01	1.00	1.00	1.00	0.55
4096 K	0.697	1.00	1.00	1.00	1.00	1.00	0.65

Table B.9: Traffic from main memory to processor cache for **sphereflake** for primary rays. Larger numbers indicate larger factors of improvement.

sphereflake — 797K triangles (97.31 MB)
primary + hard shadow rays — 117K triangles (14.34 MB) potentially visible

avail cache	frac of pot-vis	geometry traffic (MB) recursive	packet	dynamic	total traffic (MB) recursive	packet	dynamic
1 K	< 0.001	4769.71	4351.34	3587.01	4833.71	4415.34	3977.61
2 K	< 0.001	3782.92	3156.72	2141.18	3846.92	3220.72	2509.02
4 K	< 0.001	2240.22	1498.01	561.76	2304.22	1562.01	923.41
8 K	0.001	779.93	486.30	225.10	843.93	550.30	571.55
16 K	0.001	117.18	92.90	29.88	181.18	156.90	403.39
32 K	0.002	37.71	51.64	28.01	101.71	115.64	385.37
64 K	0.004	27.60	45.93	26.42	91.60	109.93	401.58
128 K	0.009	23.41	45.11	25.15	87.41	109.11	387.66
256 K	0.017	20.61	37.60	24.45	84.61	101.60	355.07
512 K	0.035	18.60	27.50	23.88	82.60	91.50	335.01
1024 K	0.070	17.00	17.66	23.52	81.00	81.66	301.68
2048 K	0.140	15.98	16.63	23.40	79.98	80.63	292.18
4096 K	0.279	15.03	15.25	21.73	79.03	79.25	270.22

avail cache	frac of pot-vis	relative improvement (geometry) recursive	packet	dynamic	relative improvement (total) recursive	packet	dynamic
1 K	< 0.001	1.00	1.10	**1.33**	1.00	1.09	**1.22**
2 K	< 0.001	1.00	1.20	**1.77**	1.00	1.19	**1.53**
4 K	< 0.001	1.00	1.50	**3.99**	1.00	1.48	**2.50**
8 K	0.001	1.00	1.60	**3.46**	1.00	1.53	1.48
16 K	0.001	1.00	1.26	**3.92**	1.00	1.15	0.45
32 K	0.002	1.00	0.73	**1.35**	1.00	0.88	0.26
64 K	0.004	1.00	0.60	**1.04**	1.00	0.83	0.23
128 K	0.009	1.00	0.52	0.93	1.00	0.80	0.23
256 K	0.017	1.00	0.55	0.84	1.00	0.83	0.24
512 K	0.035	1.00	0.68	0.78	1.00	0.90	0.25
1024 K	0.070	1.00	0.96	0.72	1.00	0.99	0.27
2048 K	0.140	1.00	0.96	0.68	1.00	0.99	0.27
4096 K	0.279	1.00	0.99	0.69	1.00	1.00	0.29

Table B.10: Traffic from main memory to processor cache for **sphereflake** for primary + hard shadow rays. Larger numbers indicate larger factors of improvement.

sphereflake — 797K triangles (97.31 MB)
primary + hard shadow + specular reflection rays
377K triangles (46.01 MB) potentially visible

avail cache	frac of pot-vis	geometry traffic (MB) recursive	packet	dynamic	total traffic (MB) recursive	packet	dynamic
1 K	< 0.001	9743.36	8943.02	7187.02	9807.36	9007.02	7622.55
2 K	< 0.001	8113.40	6783.33	4533.29	8177.40	6847.33	4946.01
4 K	< 0.001	5690.98	3868.27	1627.90	5754.98	3932.27	2034.42
8 K	< 0.001	3125.39	2020.36	838.34	3189.39	2084.36	1229.67
16 K	< 0.001	1211.01	1074.17	270.49	1275.01	1138.17	688.88
32 K	0.001	716.38	789.93	250.94	780.38	853.93	653.18
64 K	0.001	594.22	635.80	236.87	658.22	699.80	656.90
128 K	0.003	516.94	546.56	224.05	580.94	610.56	631.51
256 K	0.005	450.48	493.95	211.91	514.48	557.95	587.40
512 K	0.011	384.35	455.32	203.66	448.35	519.32	559.72
1024 K	0.022	316.83	415.70	192.86	380.83	479.70	515.96
2048 K	0.043	246.62	353.04	188.18	310.62	417.04	501.91
4096 K	0.087	177.82	231.79	177.75	241.82	295.79	471.19
avail cache	frac of pot-vis	relative improvement (geometry) recursive	packet	dynamic	relative improvement (total) recursive	packet	dynamic
1 K	< 0.001	1.00	1.09	**1.36**	1.00	1.09	**1.29**
2 K	< 0.001	1.00	1.20	**1.79**	1.00	1.19	**1.65**
4 K	< 0.001	1.00	1.47	**3.50**	1.00	1.46	**2.83**
8 K	< 0.001	1.00	1.55	**3.73**	1.00	1.53	**2.59**
16 K	< 0.001	1.00	1.13	**4.48**	1.00	1.12	**1.85**
32 K	0.001	1.00	0.91	**2.85**	1.00	0.91	**1.19**
64 K	0.001	1.00	0.93	**2.51**	1.00	0.94	1.00
128 K	0.003	1.00	0.95	**2.31**	1.00	0.95	0.92
256 K	0.005	1.00	0.91	**2.13**	1.00	0.92	0.88
512 K	0.011	1.00	0.84	**1.89**	1.00	0.86	0.80
1024 K	0.022	1.00	0.76	**1.64**	1.00	0.79	0.74
2048 K	0.043	1.00	0.70	**1.31**	1.00	0.74	0.62
4096 K	0.087	1.00	0.77	1.00	1.00	0.82	0.51

Table B.11: Traffic from main memory to processor cache for **sphereflake** for primary + hard shadow + specular reflection rays. Larger numbers indicate larger factors of improvement.

sphereflake — 797K triangles (97.31 MB)
primary + hard shadow + diffuse reflection rays
286K triangles (34.96 MB) potentially visible

avail cache	frac of pot-vis	geometry traffic (MB) recursive	packet	dynamic	total traffic (MB) recursive	packet	dynamic
1 K	< 0.001	6953.68	6864.16	5156.22	6969.68	6880.16	5457.10
2 K	< 0.001	5702.46	5544.66	3249.32	5718.46	5560.66	3544.51
4 K	< 0.001	3942.86	3679.92	1183.91	3958.86	3695.92	1477.55
8 K	< 0.001	2646.05	2399.73	612.09	2662.05	2415.73	901.92
16 K	< 0.001	1764.12	1640.35	239.04	1780.12	1656.35	535.65
32 K	0.001	1262.47	1249.98	222.08	1278.47	1265.98	514.65
64 K	0.002	927.20	943.21	208.56	943.20	959.21	505.58
128 K	0.004	681.61	695.18	196.73	697.61	711.18	490.59
256 K	0.007	490.79	508.60	185.58	506.79	524.60	471.46
512 K	0.014	347.92	379.13	176.22	363.92	395.13	457.25
1024 K	0.029	239.30	298.16	165.33	255.30	314.16	438.11
2048 K	0.057	159.13	245.75	159.20	175.13	261.75	429.64
4096 K	0.114	102.09	148.61	145.61	118.09	164.61	410.97
avail cache	frac of pot-vis	relative improvement (geometry) recursive	packet	dynamic	relative improvement (total) recursive	packet	dynamic
1 K	< 0.001	1.00	1.01	**1.35**	1.00	1.01	**1.28**
2 K	< 0.001	1.00	1.03	**1.75**	1.00	1.03	**1.61**
4 K	< 0.001	1.00	1.07	**3.33**	1.00	1.07	**2.68**
8 K	< 0.001	1.00	1.10	**4.32**	1.00	1.10	**2.95**
16 K	< 0.001	1.00	1.08	**7.38**	1.00	1.07	**3.32**
32 K	0.001	1.00	1.01	**5.68**	1.00	1.01	**2.48**
64 K	0.002	1.00	0.98	**4.45**	1.00	0.98	**1.87**
128 K	0.004	1.00	0.98	**3.46**	1.00	0.98	**1.42**
256 K	0.007	1.00	0.96	**2.64**	1.00	0.97	**1.07**
512 K	0.014	1.00	0.92	**1.97**	1.00	0.92	0.80
1024 K	0.029	1.00	0.80	**1.45**	1.00	0.81	0.58
2048 K	0.057	1.00	0.65	1.00	1.00	0.67	0.41
4096 K	0.114	1.00	0.69	0.70	1.00	0.72	0.29

Table B.12: Traffic from main memory to processor cache for **sphereflake** for primary + hard shadow + diffuse reflection rays. Larger numbers indicate larger factors of improvement.

Appendix C

Tabular Results for Distributed Memory Ray Tracer

C.1 Direct Volume Ray Casting of N-Body Particle Density

Table C.1 contains scheduler performance for direct volume ray casting on a 512-cubed particle density field. As ray calculation cost are reduced, the impact of the particular scheduling algorithm increases. When ray calculation costs are low relative to data I/O cost, our dynamic scheduling method can improve runtime by an order of magnitude over static schedules.

C.2 Ray Tracing of N-Body Particle Density Isosurface

Table C.2 contains scheduler performance for ray tracing on an isosurface of a 512^3 particle density field. The render includes two-bounce reflections and shadow rays for two directional lights. Our dynamic scheduling method can improve runtime by an order of magnitude over static schedules.

C.3 Ray Tracing of Abdominal CT Scan Isosurface

Table C.3 contains scheduler performance for ray tracing on an isosurface of a 512^3 Abdominal CT scan. The render includes shadow rays for two directional lights. When data I/O costs dominate, the domain decomposition performs on-par

with our dynamic schedule because data load costs are relatively small. Also, as the number of processes approaches the number of domains, more domains are loaded only once and remain resident throughout the execution. However, the image decomposition schedule still suffers from poor I/O efficiency even in this case, since domains are loaded on each process where rays require it. Yet, when ray costs dominate, the image decomposition performs best since the I/O costs can be amortized across many rays.

C.4 Spatial Domains Loaded from Disk

Table C.4 contains the number of domains (spatial subdivisions) loaded from disk for each schedule for each dataset. Our dynamic scheduling method significantly reduces the number of domains loaded from disk, which is the primary factor for the performance gain of our approach. A similar number of domains are accessed in our two ray cost models.

Low Ray Calculation Cost (512 x 512 image)

schedule	n=16	24	32	48	64	96	128
R-W Spread	12	8	6	4	3	3	3
Spread	26	16	13	8	7	4	4
Domain	77	53	42	29	41	20	26
Image	59	54	45	41	39	35	32

High Ray Calculation Cost (2048 x 2048 image)

schedule	n=16	24	32	48	64	96	128
R-W Spread	68	46	38	33	25	23	21
Spread	169	130	99	81	57	39	32
Domain	171	123	110	79	105	59	78
Image	118	97	76	63	57	52	44

Table C.1: Timing runs for direct volume ray casting of n-body particle density. Runtimes are in seconds per frame.

Low Ray Calculation Cost (512 x 512 image)

schedule	n=16	24	32	48	64	96	128
R-W Spread	76	56	41	35	29	22	18
Spread	204	130	91	63	50	36	28
Domain	439	390	365	278	409	218	306
Image	451	411	383	339	306	270	246

High Ray Calculation Cost (2048 x 2048 image)

schedule	n=16	24	32	48	64	96	128
R-W Spread	204	174	155	134	123	107	88
Spread	498	367	297	218	169	131	110
Domain	742	685	685	550	746	506	620
Image	742	684	645	613	566	532	494

Table C.2: Timing runs for ray tracing of n-body particle density. Runtimes are in seconds per frame.

Low Ray Calculation Cost (512 x 512 image)

schedule	n=16	24	32	48	64	96	128
R-W Spread	246	224	175	134	120	111	107
Spread	183	169	139	129	109	113	108
Domain	222	186	188	132	133	116	99
Image	400	357	419	318	281	240	240

High Ray Calculation Cost (2048 x 2048 image)

schedule	n=16	24	32	48	64	96	128
R-W Spread	1740	1321	1270	1033	847	589	591
Spread	1408	1008	1076	786	645	592	601
Domain	1133	951	1035	779	688	684	622
Image	940	690	602	487	463	336	336

Table C.3: Timing runs for ray tracing of CT scan. Runtimes are in seconds per frame.

Small N-Body Particle Density
Direct Volume Rendering

schedule	n=16	24	32	48	64	96	128
R-W Spread	1637	1483	1327	1122	795	626	639
Spread	2303	1787	1672	1417	1317	1030	762
Domain	10128	11150	11182	11507	11229	11964	11876
Image	6900	8681	10159	13392	16328	21499	26937

Small N-Body Particle Density
Isosurface + Shadows + Reflections

schedule	n=16	24	32	48	64	96	128
R-W Spread	4492	4227	4196	3959	3712	3365	2935
Spread	9281	7444	6280	5918	5829	5318	4736
Domain	29351	39069	46949	49671	64359	68910	75131
Image	26724	35201	42248	54678	65100	82620	97058

Small Abdominal CT Scan
Isosurface + Shadows

schedule	n=16	24	32	48	64	96	128
R-W Spread	183	169	139	129	109	113	108
Spread	246	224	175	134	120	111	107
Domain	222	186	188	132	133	116	99
Image	400	357	419	318	281	240	240

Table C.4: Domains loaded from disk for each scheduling strategy

Bibliography

[1] Advanced Micro Devices, Inc. AMD Opteron™ Product Data Sheet, 2007.

[2] Anant Agarwal, Mark Horowitz, and John Hennessy. An Analytical Cache Model. *ACM Transactions on Computer Systems*, 7(2):184–215, May 1989.

[3] Timo Aila and Tero Karras. Architecture Considerations for Tracing Incoherent Rays. In M. Doggett, S. Laine, and W. Hunt, editors, *Proceedings of High Performance Graphics*, 2010.

[4] Timo Aila and Samuli Laine. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of High Performance Graphics*, 2009.

[5] John Amanatides. Ray Tracing with Cones. *Computer Graphics*, 18(3):129–135, July 1984.

[6] John Amanatides and Andrew Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Proceedings of Eurographics ́87*, pages 1–10, August 1987.

[7] Jennifer M. Anderson, Saman P. Amarasinghe, and Monica S. Lam. Data and Computation Transformations for Multiprocessors. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[8] Jennifer M. Anderson and Monica S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 112–125, June 1993.

[9] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conference*, volume 32, pages 37–45, 1968.

[10] Sigal Ar, Gil Montag, and Ayellet Tal. Deferred, Self-Organizing BSP Trees. In *Eurographics 2002*, 2002.

[11] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithm and Architectures*, pages 119–129, 1998.

[12] James Arvo and David Kirk. Fast ray tracing by ray classification. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, volume 21, pages 55–64. Association for Computing Machinery, 1987.

[13] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich. Ray Tracing on the CELL Processor. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 25–23, 2006.

[14] Carsten Benthin, Ingo Wald, and Philipp Slusallek. A Scalable Approach to Interactive Global Illumination. *Computer Graphics Forum (Proceedings of Eurographics 2003)*, 22(3):621–630, 2003.

[15] Tony Bernardin, Brian Budge, and Bernd Hamann. Stack-Based Visualization of Out-of-Core Algorithms. In *Proceedings of ACM SoftVis*, 2008.

[16] James Bigler, Abe Stephens, and Steven Parker. Design for Parallel Interactive Ray Tracing Systems. In *Proceedings of Interactive Ray Tracing*, 2006.

[17] Solomon Boulos, Dave Edwards, J Dylan Lacewell, Joe Kniss, Jan Kautz, Ingo Wald, and Peter Shirley. Packet-Based Whitted and Distribution Ray Tracing. In *Proceedings of Graphics Interface 2007*, 2007.

[18] Solomon Boulos, Ingo Wald, and Carsten Benthin. Adaptive Ray Packet Reordering. In *Proceedings of Interactive Ray Tracing*, 2008.

[19] Brian Budge, Tony Bernardin, Jeff A. Stuart, Shubhabrata Sengupta, Kenneth I. Joy, and John D. Owens. Out-of-Core Data Management for Path Tracing on Hybrid Resources. In P. Dutré and M. Stamminger, editors, *Proceedings of Eurographics*, 2009.

[20] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler Optimizations for Improving Data Locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, October 1994.

[21] Hank Childs, Eric S. Brugger, Kathleen Bonnell, Jeremy S. Meredith, Mark Miller, Brad J. Whitlock, and Nelson Max. A Contract-Based System for Large Data Visualization. In *Proceedings of IEEE Visualization*, pages 190–198, 2005.

[22] Hank Childs, Mark A. Duchaineau, and Kwan-Liu Ma. A Scalable, Hybrid Scheme for Volume Rendering Massive Data Sets. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 153–162, 2006.

[23] Hank Childs, David Pugmire, Sean Ahern, Brad Whitlock, Mark Howison, Prabhat, Gunther Weber, and E. Wes Bethel. Extreme Scaling of Production Visualization Software on Diverse Architectures. *IEEE Computer Graphics and Applications*, pages 22–31, 2010.

[24] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed Ray Tracing. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, pages 137–145, July 1984.

[25] Frank Dachille IX and Arie Kaufman. GI-Cube: An Architecture for Volumetric Global Illumination and Rendering. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 119–128, August 2000.

[26] David E. DeMarle, Christiaan P. Gribble, Solomon Boulos, and Steven G. Parker. Memory Sharing for Interactive Ray Tracing on Clusters. *Parallel Computing*, 31(2):221–242, February 2005.

[27] David E. DeMarle, Christiaan P. Gribble, and Steven G. Parker. Memory-Savvy Distributed Interactive Ray Tracing. In Dirk Bartz, Bruno Raffin, and Han-Wei Shen, editors, *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, 2004.

[28] David E. DeMarle, Steven Parker, Mark Hartner, Christiaan Gribble, and Charles Hansen. Distributed Interactive Ray Tracing for Large Volume Visualization. In *Proceedings of the Symposium on Parallel and Large-Data Visualization and Graphics*, 2003.

[29] Mark Dippé and John Swensen. An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis. *Computer Graphics (Proceedings of SIGGRAPH 1984)*, 18(3):149–158, July 1984.

[30] Peter Djeu, Warren Hunt, Rui Wang, Ikrima Elhassan, Gordon Stoll, and William R. Mark. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. *ACM Transactions on Graphics (conditionally accepted)*, 2007.

[31] Jan Edler and Mark D. Hill. Dinero IV cache simulator. Technical report, University of Wisconsin, 1998.

[32] Arno Formella, Christian Gill, and Volker Hofmeyer. Fast Ray Tracing of Sequences by Ray History Evaluation. In *Proceedings of Computer Animation 1994*, pages 184–191. IEEE Computer Society Press, May 1994.

[33] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On Visible Surface Generation by A Priori Tree Structures. In *Proceedings of the 7th An-*

nual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), pages 124–133, 1980.

[34] Donald S. Fussell and K. R. Subramanian. Fast Ray Tracing Using k-D Trees. Technical Report CS-TR-88-07, The University of Texas at Austin, 1, 1988.

[35] Stuart A. Green and Derek J. Paddon. Exploiting Coherence for Multiprocessor Ray Tracing. *IEEE Computer Graphics and Applications*, 9(11):12–26, 1989.

[36] Stuart A. Green and Derek J. Paddon. A Highly Flexible Multiprocessor Solution for Ray Tracing. *The Visual Computer*, 6:62–73, 1990.

[37] Christiaan P. Gribble and Steven G. Parker. Enhancing Interactive Particle Visualization with Advanced Shading Models. In *Proceedings of the 3rd Symposium on Applied Perception in Graphics and Visualization*, pages 111–118, 2006.

[38] Eric A. Haines. A Proposal for Standard Graphics Environments. *IEEE Computer Graphics & Applications*, 7(11):3–5, 1987.

[39] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, 2000.

[40] V. Havran, J. Prikryl, and W. Purgathofer. Statistical Comparison of Ray-Shooting Efficiency Schemes. Technical report, Vienna University of Technology, April 2000.

[41] Paul S. Heckbert. Ray Tracing JELL-O® Brand Gelatin. *Computer Graphics (Proceedings of SIGGRAPH)*, 21(4):73–74, July 1987.

[42] Paul S. Heckbert. Ray Tracing JELL-O® Brand Gelatin. *Communications of the ACM*, 31(2):131–134, February 1988.

[43] Paul S. Heckbert and Pat Hanrahan. Beam Tracking Polygonal Objects. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 119–127, 1984.

[44] Mark Howison, E.Wes Bethel, and Hank Childs. MPI-Hybrid Parallelism for Volume Rendering on Large, Multi-Core Systems. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2010.

[45] Warren Hunt, William R. Mark, and Gordon Stoll. Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. In *IEEE Symposium on Interactive Ray Tracing*, 2006.

[46] Warren A. Hunt. *Data Structures and Algorithms for Real-Time Ray Tracing at the University of Texas at Austin*. PhD thesis, The University of Texas at Austin, December 2008.

[47] Jim Hurley, Alexander Kapustin, Alexander Reshetov, and Alexei Soupikov. Fast Ray Tracing for Modern General Purpose CPU. In *Graphicon 2002*, 2002.

[48] Homan Igehy. Tracing Ray Differentials. In *Proceedings of SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pages 179–186, August 1999.

[49] Ilian Iliev and Paul Shapiro. private communication, 2009.

[50] Intel Corporation. Intel® 5000X Chipset Overview, 2007.

[51] Intel Corporation. Intel® 975X Express Chipset, 2007.

[52] H. V. Jagadish. Linear Clustering of Objects with Multiple Attributes. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 332–342. ACM, ACM Press, 1990.

[53] Henrik Wann Jensen. Global Illumination using Photon Maps. In *Eurographics Rendering Workshop 1996*, pages 21–30, 1996.

[54] Henrik Wann Jensen and Per H. Christensen. Efficient Simulation of Light Transport in Scenes With Participating Media Using Photon Maps. In *Proceedings of SIGGRAPH*, pages 311–320, 1998.

[55] David Jevans and Brian Wyvill. Adaptive Voxel Subdivision for Ray Tracing. In *Proceedings of Graphics Interface 1989*, pages 164–172. Canadian Information Processing Society, June 1989.

[56] James T. Kajiya. The Rendering Equation. *Computer Graphics (Proceedings of SIGGRAPH)*, 20:143–150, August 1986.

[57] Toshi Kato. ”Kilauea” – Parallel Global Illumination Renderer. *Parallel Computing*, 29:289–310, 2003.

[58] Toshi Kato and Jun Saito. ”Kilauea” – Parallel Global Illumination Renderer. In D. Bartz, X. Pueyo, and E. Reinhard, editors, *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization (EGPGV)*, pages 7–16, 2002.

[59] Hiroaki Kobayashi, Satoshi Nishimura, Hideyuki Kubota, Tadao Nakamura, and Yoshiharu Shigei. Load Balancing Strategies for a Parallel Ray-Tracing System Based on Constant Subdivision. *The Visual Computer*, 4:197–209, 1988.

[60] Craig Kolb, Don Mitchell, and Pat Hanrahan. A Realistic Camera Model for Computer Graphics. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 317–324, 1995.

[61] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: a 32-way multithreaded SPARC processor. In *IEEE MICRO 2005*, volume 25, pages 21–29, 2005.

[62] Samuli Laine. Restart Trail for Stackless BVH Traversal. In M. Doggett, S. Laine, and W. Hunt, editors, *Proceedings of High Performance Graphics*, 2010.

[63] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 63–74, New York, NY, USA, 1991. ACM Press.

[64] Wilfrid Lefer. An Efficient Parallel Ray Tracing Scheme for Distributed Memory Parallel Computers. In *Proceedings of the Parallel Rendering Symposium*, pages 77–80, 1993.

[65] Marc Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[66] Marc Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[67] Jonas Lext and Thomas Akenine-Möller. Towards Rapid Reconstruction for Animated Ray Tracing. Short Presentation — Animation and Physical Modelling, Eurographics 2001, 2001.

[68] Jonas Lext, Ulf Assarsson, and Thomas Akenine-Möller. BART: A Benchmark for Animated Ray Tracing. Technical report, Department of Computer Engineering, Chalmers University of Technology, May 2000. Available at http://www.ce.chalmers.se/BART/.

[69] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning. In *ACM*

SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 102–112, 2001.

[70] William R. Mark and Donald Fussell. Real-Time Rendering Systems in 2010. Technical Report TR-05-18, The University of Texas at Austin, May 2005.

[71] Gerd Marmitt, Heiko Friedrich, and Philipp Slusallek. Interactive Volume Rendering with Ray Tracing. In *Proceedings of EUROGRAPHICS STAR — State of The Art Report*, 2006.

[72] Joel McCormack, Robert McNamara, Christopher Gianos, Larry Seiler, Norman P. Jouppi, and Ken Correll. Neon: A Single-Chip 3D Workstation Graphics Accelerator. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 123–132. ACM, ACM Press, 1998.

[73] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4), July 1996.

[74] Bongki Moon, H.V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the Clustering Properties of Hilbert Space-filling Curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, January / February 2001.

[75] Andrew Naiberg. Overcoming the I/O Bottleneck with General Parallel File System. *IBM Systems Magazine*, June 2005.

[76] Paul A. Navrátil, Donald S. Fussell, Calvin Lin, and William R. Mark. Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization. In *Proceedings of Interactive Ray Tracing*, 2007.

[77] Paul A. Navrátil and William R. Mark. An Analysis of Raytracing Bandwidth Comsumption. Technical Report TR-06-40, The University of Texas at Austin, Department of Computer Sciences, 2006.

[78] Steven Parker, Michael Parker, Yaren Livnat, Peter Pike Sloan, Chuck Hansen, and Peter Shirley. Interactive Ray Tracing for Volume Visualization. *IEEE Transactions on Computer Graphics and Visualization*, 5(3):238–250, July-September 1999.

[79] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive Ray Tracing for Isosurface Rendering. In *Proceedings of IEEE Visualization*, pages 233–238, 1998.

[80] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive Ray Tracing. In *Proceedings of Interactive 3D Graphics*, 1999.

[81] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. OptiX: a General-Purpose Ray Tracing Engine. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 29:66, July 2010.

[82] Tom Peterka, Hongfeng Yu, Robert Ross, Kwan-Liu Ma, and Rob Latham. End-to-End Study of Parallel Volume Rendering on the IBM Blue Gene/P. In *Proceedings of International Conference on Parallel Processing*, pages 566–573, 2009.

[83] Matt Pharr and Pat Hanrahan. Geometry Caching for Ray-Tracing Displacement Maps. In Xavier Pueyo and Peter Schröder, editors, *Eurographics Rendering Workshop 1996*, pages 31–40, New York City, NY, 1996. Springer Wien.

[84] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 2004.

[85] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. *Computer Graphics (Proceedings of SIGGRAPH)*, 31(Annual Conference Series):101–108, August 1997.

[86] James Philbin, Jan Edler, Otto J. Anshus, Craig C. Douglas, and Kai Li. Thread Scheduling for Cache Locality. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 60–71, 1996.

[87] Michael Potmesil and Indranil Chakravarty. A Lens and Aperture Camera Model for Synthetic Image Generation. In *Proceedings of the 8th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 297–305, 1981.

[88] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002.

[89] Erik Reinhard, Alan G. Chalmers, and Frederik W. Jansen. Hybrid Scheduling for Parallel Rendering using Coherent Ray Tasks. In *Proceedings of IEEE Parallel Visualization and Graphics Symposium*, 1999.

[90] Erik Reinhard and Frederik W. Jansen. Rendering Large Scenes using Parallel Ray Tracing. In *Proceedings of Eurographics Workshop of Parallel Graphics and Visualization*, 1996.

[91] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 24(3):1176–1185, 2005.

[92] Steven M. Rubin and Turner Whitted. A 3-dimensional Representation for Fast Rendering of Complex Scenes. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 110–116, 1980.

[93] John Salmon and Jeff Goldsmith. A Hypercube Ray-Tracer. In G.C. Fox, editor, *Proceedings of the Third Conference on Hypercube Computers and Applications*, pages 1194–1206, 1988.

[94] Issac D. Scherson and Elisha Caspary. Multiprocessing for Ray Tracing:

a Hierarchical Self-Balancing Approach. *The Visual Computer*, 4(4):188–196, July 1988.

[95] Jörg Schmittler. *SaarCOR: A hardware-Architecture for Realtime Ray Tracing*. PhD thesis, Computer Graphics Group, Saarland University, 2006.

[96] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR - A Hardware Achitecture for Ray Tracing. In *Proceedings of Eurographics Workshop on Graphics Hardware*, pages 27–36. European Association for Computer Graphics, September 2002.

[97] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Graphics Hardware*, pages 95–106, 2004.

[98] SimpleScalar. SimpleScalar LLC (http://www.simplescalar.com/).

[99] Joshua Steinhurst, Greg Coombe, and Anselmo Lastra. Reordering for Cache Conscious Photon Mapping. In *Proceedings of Graphics Interface*, pages 97–104, 2005.

[100] Gordon Stoll. Parallel & Distributed Processing. In *ACM SIGGRAPH 2005 Courses*, 2005.

[101] Gordon Stoll, William R. Mark, Peter Djeu, Rui Wang, and Ikrima Elhassan. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. Technical Report TR-06-21, The University of Texas at Austin, 2006.

[102] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A Characterization of Ten Hidden-Surface Algorithms. *ACM Computing Surveys*, 6(1):1–55, March 1974.

[103] TOP500.org. Architecture Share for 06/2010, June 2010.

[104] Douglas Voorhies. *Graphics Gems 2*, chapter Space-Filling Curves and a Measure of Coherence, pages 26–30, 485–486. Academic Press, 1991.

[105] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[106] Ingo Wald, Carsten Benthin, and Philipp Slusallek. A Simple and Practical Method for Interactive Ray Tracing of Dynamic Scenes. Technical report, Saarland University, 2002.

[107] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 2003.

[108] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Interactive Global Illumination in Complex and Highly Occluded Environments. In *Proceedings of the 14th Eurographics Workshop on Rendering*, 2003.

[109] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1):6, 2007.

[110] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray Tracing Animated Scenes Using Coherent Grid Traversal. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 25(3):485–493, 2006.

[111] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive Global Illumination using Fast Ray Tracing. In *Proceedings of the 13th Eurographics Workshop on Rendering*, 2002.

[112] Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley. State of the Art in Ray Tracing Animated Scenes. In *Proceedings of EUROGRAPHICS STAR — State of The Art Report*, 2007.

[113] Ingo Wald, Timothy J. Purcell, Jörg Schmittler, Carsten Benthin, and Philipp Slusallek. Realtime Ray Tracing and Its Use for Interactive Global Illumination. In *Proceedings of EUROGRAPHICS STAR — State of The Art Report*, 2003.

[114] Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, pages 277–288, 2001.

[115] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum (Proceedings of Eurographics 2001)*, 20(3):153–164, 2001.

[116] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D.P. Greenberg. Lightcuts: a Scalable Approach to Illumination. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 24(3):1098–1107, 2005.

[117] Bruce Walter, Adam Arbree, Kavita Bala, and Donald P. Greenberg. Multidimensional Lightcuts. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 25(3):1081–1088, 2006.

[118] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved Computational Methods for Ray Tracing. *ACM Transactions on Graphics*, 3(1):52–69, January 1984.

[119] Turner Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 6(23):343–349, June 1980.

[120] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. *Transactions on Parallel and Distributed Systems*, October 1991.

[121] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: a Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 24(3):434–444, 2005.

Vita

Paul Arthur Navrátil was raised in Houston, Texas and graduated from Bellaire High School in 1994. He received a Bachelor of Science degree with honors in Computer Science and a Bachelor of Arts degree with honors in Plan II Interdisciplinary Honors from the University of Texas at Austin in May, 1999; he graduated Phi Beta Kappa and was recognized as a Dean's Distinguished Graduate in the College of Liberal Arts. In January 1999, he began work at Liaison Technology, a data mining start-up company based in Austin, Texas. He returned to the University of Texas at Austin for graduate study in January, 2001, and he received a Master of Science degree in Computer Science in May, 2006.

Permanent address: 7514 Dallas Drive
Austin, Texas 78729

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.