Appears in 41^{st} *Annual IEEE/ACM International Symposium on Microarchitecture, 2008*

Toward A Multicore Architecture for Real-time Ray-tracing

Venkatraman Govindaraju*, Peter Djeu†, Karthikeyan Sankaralingam*, Mary Vernon*, William R. Mark†
*Department of Computer Sciences
University of Wisconsin-Madison
†Department of Computer Sciences
The University of Texas at Austin
Vertical Research Group (vertical@cs.wisc.edu)

Abstract

Significant improvement to visual quality for real-time 3D graphics requires modeling of complex illumination effects like soft-shadows, reflections, and diffuse lighting interactions. The conventional Z-buffer algorithm driven GPU model does not provide sufficient support for this improvement. This paper targets the entire graphics system stack and demonstrates algorithms, a software architecture, and a hardware architecture for real-time rendering with a paradigm shift to ray-tracing. The three unique features of our system called Copernicus are support for ***dynamic scenes, high image quality,*** *and* ***execution on programmable multicore architectures****. The focus of this paper is the synergy and interaction between applications, architecture, and evaluation. First, we describe the ray-tracing algorithms which are designed to use redundancy and partitioning to achieve locality. Second, we describe the architecture which uses ISA specialization, multithreading to hide memory delays and supports only local coherence. Finally, we develop an analytical performance model for our 128-core system, using measurements from simulation and a scaled-down prototype system. More generally, this paper addresses an important issue of mechanisms and evaluation for challenging workloads for future processors. Our results show that a single 8-core tile (each core 4-way multithreaded) can be almost 100% utilized and sustain 10 million rays/second. Sixteen such tiles, which can fit on a* $240mm^2$ *chip in 22nm technology, make up the system and with our anticipated improvements in algorithms, can sustain real-time rendering. The mechanisms and the architecture can potentially support other domains like irregular scientific computations and physics computations.*

1. Introduction

Providing significant improvement to visual quality for real-time 3D graphics requires a system-level rethink from algorithms down to the architecture. Conventional real-time 3D graphics is specialized to support a single graphics rendering algorithm: the Z-buffer. This model is implemented with a fixed pipeline and some programmable components. The programs are restricted in how they communicate with each other and, if at all, with global memory. This *Ptolemic* universe of algorithms, software, and architecture, revolves around the Z-buffer, and has provided great success thus far.

Future visual quality improvements in real-time 3D graphics require more realistic modeling of lighting and complex illumination effects, which cannot be supported by the Z-buffer algorithm. Among other things, these effects require visibility computations (light-ray and surface intersection) involving *rays with variety of origins and directions*, while the Z-buffer is optimized for *regularly spaced rays originating from a single point*.

Modern rendering systems live in the *Ptolemic* Z-buffer universe and provide some support for these effects by over-extending the use of the Z-buffer algorithm. This results in several problems, namely, lack of algorithmic robustness, poor artist and programmer productivity, system complexity for programmers and severe constraints on the artist and types of scenes. The recently announced Larrabee architecture takes a step outside this *Ptolemic* universe by implementing the special-purpose Z-buffer hardware using customizable high-performance software running on a flexible parallel architecture [30]. Product reality, backward compatibility and industry's apprehension to disruptive change force these steps to be gradual.

Graphics algorithms and VLSI technology have reached a point where a paradigm shift to a *Copernican* universe centered around applications and sophisticated visibility algorithms with ray-tracing is possible. Ray-tracing naturally supports realistic lighting and complex illumination and can thus provide the next generation of visual quality. The *architectural challenge* in a ray-tracing system is providing a highly parallel and scalable architecture that can efficiently support data-parallel computation and irregular read-modify-write memory accesses. The *software and algorithm challenge* is to develop a ray-tracing system that is highly parallel, exploits locality, and reduces synchronization. The *evaluation challenge* is to accurately project performance with design trade-offs for a complex new application on a new architecture.

This paper starts this paradigm shift by presenting our full

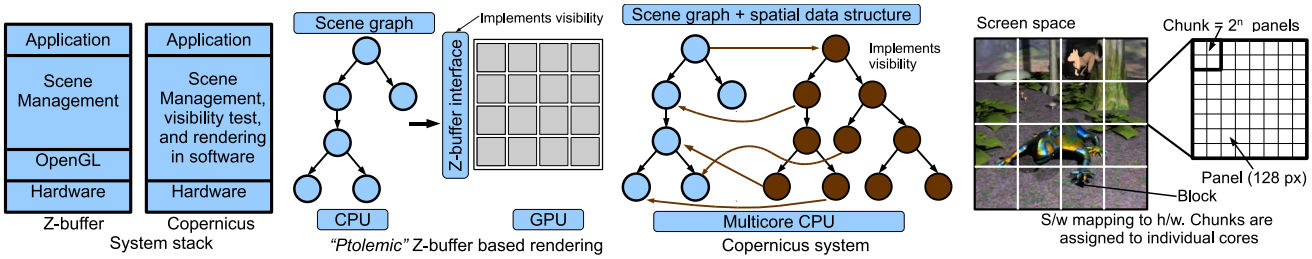


Figure 1. Copernicus system overview and comparison to Z-buffer

system design called Copernicus, outlined in Figure 1. Copernicus includes several co-designed hardware and software innovations. Razor, the software component of Copernicus, is a highly parallel, multigranular (at the coarse grain it performs domain decomposition and in the fine grain it traces multiple rays), locality-aware ray tracer. The hardware architecture is a large-scale tiled multicore processor with private L2 caches only, fine-grained ISA specialization tuned to the workload, multi-threaded for hiding memory access latency and local cache coherence. In addition to providing a new level of image quality, the application-driven approach and programmability in Copernicus allows multi-way tradeoffs between performance, image quality, image size, and frames-per-second. In a game-specific environment, collision detection, approximate physics simulations and scene management can also be performed by the rendering system.

To evaluate Copernicus, we developed a customized analytic model seeded with inputs and validated by a scaled prototype and a full system simulator. This model captures the impact of multi-threaded execution and memory access conflicts. We estimate a 16-tile chip, with 8 cores in a tile, can fit on $240mm^2$ chip at 22nm technology. When running at 4GHz, this system can sustain real-time frame-rates for realistic benchmarks, rendering up to 74 million rays per second over all, and 10 million rays per second in a tile.

The four main contributions of this paper are: 1) the first complete full system single-chip design for high quality real-time rendering of dynamic scenes using ray-tracing with effects like soft shadows; 2) an architecture that is a realistic transformation path for graphics hardware from "specialized" embarrassingly-parallel architectures to flexible high-throughput parallel architectures; 3) a detailed characterization and analysis framework for a future computation workload; 4) and a customized analytic model that captures the impact of multi-threading and memory access conflicts.

The rest of this paper is organized as follows. Section 2 describes our algorithms and software architecture. Section 3 presents a detailed characterization of this workload and Section 4 describes our architecture design. Section 5 presents our performance results and Section 6 discusses related work. Section 7 discusses synergy between application, architecture, and evaluation, and extensions beyond graphics, and Section 8 concludes.

2. Graphics algorithms and Razor

One of the big challenges for future architectures is the architecture/applications "chicken and egg" problem. To study architectural tradeoffs, we need an application optimized for that architecture. For many consumer applications, future workloads may not exist yet for *any* architecture, because additional performance is used to increase quality, rather than reduce time to solution. To overcome this "chicken and egg" problem for our ray-tracing based system, we explicitly designed to explore *future* graphics capabilities on *future* multicore architectures. For a full discussion of the graphics algorithms refer to [5]. Below, we provide an overview emphasizing the workload and architecture interaction.

Basics: Ray-tracing is a technique for generating images by simulating the behavior of light within a 3D scene by typically tracing light rays from the camera into the scene [25]. In general two types of rays are used. *Primary rays* are traced from a particular point on the camera image plane (a pixel) into the scene, until they hit a surface, at a so-called *hit point*. *Secondary rays* are traced from a hit point to determine how it is lit. Finally, to determine how the surface material appears *texture lookups* and *shading computations* are performed at or near the hit point.

Razor description: Our ray-tracing system, called Razor supports dynamic scenes (i.e. scenes that change geometry with user interaction like breaking a wall in a game) with high quality effects like soft shadows and defocus blurring using a general visibility model. As shown in Figure 1, it implements visibility using a spatial data structure (specifically a kd-tree) built for every frame, unlike a traditional ray-tracer which does this once as part of pre-processing. Razor uses coarse-grain concurrency by domain decomposition of frames into "chunks", and fine-grained concurrency by packet-tracing many rays simultaneously as shown in Figure 1. Material shading computations are decoupled from intersection computations since they differ significantly.

Razor analysis: The *spatial data structure (kd-tree) and per-frame rebuild* allows objects to move in the scene. However, this requires fine granularity read-modify-write support in the memory. The explicit *chunk-based* concurrency in Razor allows each processor to build its own copy of the spatial data structure, avoiding global synchronization. The storage overhead of "replication" is relatively small because the

primary rays assigned to different chunks access mostly non-overlapping parts of the data structure. Secondary rays could disrupt this behavior, but empirically we found their impact to be small. Decoupling material shading from intersection and fine-grained packet-tracing provides fine-grain data-parallelism.

From analysis of the algorithm, we derive the hardware challenges for our Copernicus system: (a) Large computation demand requiring a highly parallel architecture, (b) Key data structures are modified or rebuilt every frame, with this work done on-demand rather than in a temporally separate phase, (c) These data structures are irregular, and thus difficult to support with software managed caches, (d) The size of the key data structures is large (on the order of a few gigabytes), so the architecture must exploit locality.

Sofware implementation: To drive the algorithms and architectural exploration we developed a fully functional implementation of Razor. Razor is parallelized using multi-threading, with one main thread and worker threads for each processor core. Primary rays are organized into 2D screen-space "chunks" which are placed onto a work queue and mapped to worker threads as shown in Figure 1. Chunks are sized to capture significant memory-access locality without disrupting load-balance. We optimized Razor using SSE intrinsics with the Intel Compiler running on Windows, to drive the algorithm development. For architecture studies, we implemented a flexible version of Razor that compiles on Gcc/Unix which we have tested on x86/Linux and Sparc/Solaris. It uses our cross-platform SSE library that implements the intrinsics using C with the native compiler generating instructions available in the machine. While Razor is designed for future hardware, it runs on today's hardware at 0.2 to 1.1 frames/sec on a commodity 8-core (2 socket x quad-core) system and is robust enough to support scenes extracted from commercial games.

Related work: Ray-tracing has been studied for a long time [31], [39]. Only recently researchers have begun to examine its potential for interactive rendering with systems targeted at today's hardware [23], [1], [9], ASICs [33], [35], [40], FPGAs [36], or distributed clusters [28], [38]. In general, these systems are not representative of future systems in one or more of the following ways: They do not address the parallel scalability of single-chip manycore hardware; they compromise image quality and functionality so that the workload is simpler than that of future real-time ray-tracing workloads; or the workload is restricted to the easier case of static or rigid-body scenes. The Manta system [2] is a notable partial exception – its scalability on large multi-chip SGI shared memory machines has been studied.

3. Workload characterization

This section presents a detailed characterization of our rendering system, focusing on the parallelism, memory behavior, computation, and sensitivity to input. We analyze both the unoptimized and x86-optimized implementations to drive our architecture design. Unlike simulation-only characterization or back-of-envelope estimates, we use performance counter measurements on existing systems to motivate newer systems. Such a methodology is essential to break out of the "chicken and egg" application/architecture problem.

Infrastructure: For performance tuning and application development we used two scaled-prototype systems: namely a dual-socket quad-core Intel processor (Clovertown - Intel Xeon E5345 2.33GHz) that allows us to examine 8-way scalability and a Niagara system (Sun-Fire-T200) that allows us to examine 32-way scalability. In addition, these two systems allow us to compare out-of-order processing to in-order processing, and the benefits of simultaneous multi-threading. We used PAPI-based performance counter measurement [20], gprof [7], Valgrind [21], and Pin [26] to characterize Razor. In addition, we used the Multifacet GEMS simulation [18] environment for sensitivity studies. Table 1 briefly characterizes our benchmark scenes which are from game and game-like scenes and representative of real workloads.

3.1. Parallelism

Figure 2 characterizes Razor's parallelism behavior using our prototype systems and shows it has abundant parallelism. Figure 2a shows performance on a single core (in-order Niagara) as more threads are provided. We can see near-linear speedup going from 1 to 4 threads. Our analytical model of processor utilization (details in section 5.2) and simulation results show that at 1-thread the processor is only 20% utilized and hence executing 4 threads provides speedup. This parallel speedup shows that the working set size of each thread is small enough to fit in the L2 cache and multithreading hides the latency of the L1 misses.

Figure 2b and 2c show parallel scalability as the amount of resources (# cores) is increased from 1 core to 8 cores: speedups are relative to the baseline 1-core. Niagara running in this configuration under-utilizes the cores because only one thread is assigned to each core. Razor exhibits near-linear speedups, and a 16-thread configuration with two threads per core shows similar behavior. The Clovertown system shows linear speedup up to 4 cores and beyond that its front-side bus based communication hinders parallel speedups. Niagara speedups for this application require some explanation, since it has one FPU per chip. Our SSE library when executed natively on a SPARC machine like Niagara does not have access to real SIMD registers. Due to gcc's aliasing analysis, these variables are not register-allocated and instead are in memory. Hence, every intrinsic call, even though inlined results in 8 memory instructions at least, thus making the FPU under-utilized. On our simulator, we effectively emulate SIMD registers and operations which are incorporated into our instruction mix analysis and performance estimation.

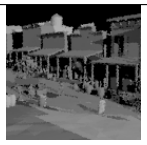
						
1)	Benchmark	Courtyard	Fairyforest	Forest	Juarez	Saloon
2)	Complexity (# triangles)	31,393	174,117	54,650	154,789	172,249
3)	Memory reserved (MB)	2900	400	2600	1400	1100
4)	Memory used dynamically (MB)	559	122	515	501	236
5)	Estimate of maximum bandwidth required for real-time (GB/s)	13.7	2.9	12.7	12.2	5.7
6)	Unique memory references per 100 million instructions	20482	15976	22312	83264	73650
7)	Rendering time (s)	6.3	9.2	12.9	17.5	10.5
8)	Rendering time - Fast setting (s)	1.1	0.8	0.3	0.2	0.4
9)	Description	Skinned and animated characters	Benchmark scene	Catmull-Clark control mesh, skinned and animated	*Call of Juarez* outdoors	*Call of Juarez* indoors
Application profiling results from gprof. Percentage contribution of different functions						
10)	Functions					
11)	UnifiedKDTreeIntersectAscending	34.9%	32.6%	35.5%	28.7%	26.7%
12)	rzRayBoxIsectOne	17.4%	10.9%	19.3%	29.0%	25.0%
13)	segmentOneFootprintIncrQ	18.6%	12.1%	9.4%	7.7%	9.0%
14)	IntersectRayKDLeafP	7.9%	6.5%	9.8%	12.5%	10.5%
15)	rzComputeIsectTriPair	3.8%	7.6%	1.8%	6.5%	6.3%
16)	ProjectAndLerpOneVertex	3.9%	4.8%	3.3%	5.6%	6.1%
17)	QuadSample	7.7%	8.4%	3.7%	2.9%	4.6%
18)	Other	5.9%	17.1%	17.3%	7.1%	11.9%

Table 1. Benchmark scenes. All scenes (except "fast settings") rendered at 1024x1024 image size, and contain 2 lights, 8 primary rays/pixel, 4 secondary rays per primary-ray (40 rays/pixel total). Rendering times are measured on Clovertown system with eight threads. Rendering times on Niagara are about 12X worse - due to in-order processing, no SSE, and lower frequency

In these experiments only computation resources are scaled while memory bandwidth, capacity, and caches remain fixed[1]. These results show that Razor is clearly well balanced and highly parallel.

Observation 1: Razor's physically based optimizations that use redundancy to obtain parallelism work well.

Observation 2: There is abundant parallelism in Razor.

3.2. Memory

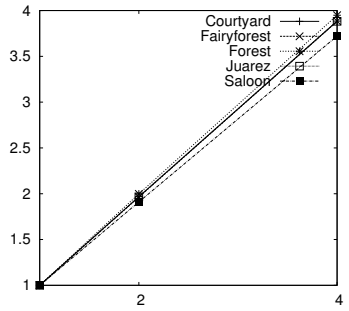
One of the key insights in the structuring of Razor is to exploit coherence in primary rays (i.e. rays with physical proximity touch physically close-by objects, and thus exhibit locality in the spatial data structure) and secondary rays, which although somewhat less coherent still have reasonable locality in the data structures they need to access. As a result, redundantly building the kd-tree for every thread is not expected to result in linear memory growth. To study this behavior, we track memory usage within the application and vary the number of threads from 1 to 32. Figure 3a shows this memory footprint growth which is quite slow and quickly tapers off. Except for one scene, going from 1 to 32 threads increases memory by less than 2X. For `fairyforest`, the memory consumption grows by 5X because this scene has an extremely finely tessellated surface which is represented with a very flat hierarchy, resulting in high duplication overhead. The scene graph can be reconstructed to avoid such duplication. Overall, this shows that the common case assumptions that Razor is based on are valid and work. Table 1 (row 3) shows the actual memory consumed for the baseline 1-thread configuration.

Observation 3: Per-thread duplication of the spatial data structures results in little memory overhead.

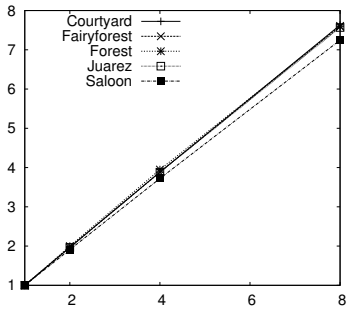
We examine detailed memory behavior in several ways. First, to determine working set size, we simulated Razor in a machine configuration with a single level of a fully associative cache and vary the cache size from 4KB to 256MB as shown in Figure 3b. The miss rate rapidly tapers off at 64KB which shows the working set is quite small even though the total memory usage is of order of GB (Table 1 row three and four). In these experiments we render only one frame of the scene. If we rendered multiple frames and set the cache to match the memory footprint, these "compulsory" misses will also go away. However, Razor rebuilds its spatial data structures for every frame to account for dynamic effects, and since the amount of computation is quite large for a full frame, cache reuse across frames is immaterial.

To quantify locality and measure the instantaneous working set, we sampled the application every 100 million instructions and measured the total number of memory references

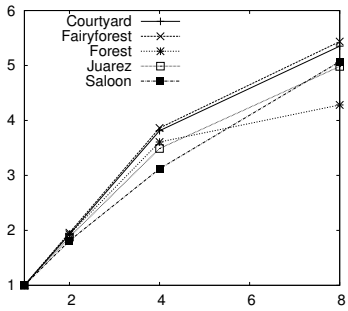
1. On the Clovertown prototype, cache doubles when going from 4 to 8 cores, and memory bandwidth by a smaller amount, as the two chips communicate through the front-side bus.



(a) Niagara 1-core with multiple threads per core

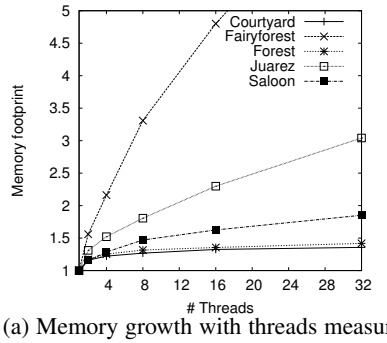


(b) Niagara multiple cores (1 thread per core)

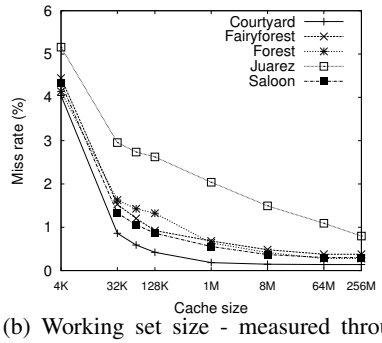


(c) Clovertown multiple cores (1 thread per core)

Figure 2. Razor parallelism. For all graphs X-axis is number of threads and Y-axis is speedup.



(a) Memory growth with threads measured on Clovertown. (Same trend on Niagara)



(b) Working set size - measured through simulation

Benchmark	L1 miss rate %	L2 miss rate %
Courtyard	1.4	2.2
Fairyforest	1.5	4.4
Forest	1.9	5.7
Juarez	1.5	26.1
Saloon	1.5	5.8

(c) Locality measured from simulating Niagara-like configuration.

Figure 3. Razor memory behavior.

and unique memory references using Pin. The sixth row in Table 1 shows the average unique memory references per sample. Across all the scenes, the number of unique addresses (measured at cache line size granularity) is between 15000 and 80000 for 100 million instructions, while the total references is between 45 million and 50 million. This shows high spatial locality in the kd-tree is indeed achieved by the coherence between primary rays and the localized nature of secondary rays. Figure 3c shows our measured L1 and L2 data cache miss rates derived from simulating a Niagara-like configuration. Since the working set is quite small, we can see that the miss rates are quite low.

Observation 4: The instantaneous working set of the application is quite small.

Observation 5: The data traversed is scene dependent and is fundamentally recursive and hence static streaming techniques cannot capture the working set.

3.3. Computation

Table 2 describes the basic computation characteristics. Columns 2 through 6 show the breakdown of computation in terms of different functional units used. The code is very memory intensive with only a small fraction of branches. However, most of these branches are data-dependent and hence cannot be supported efficiently with just predication. We measured branch prediction accuracy in our Clovertown prototype and found it to be greater than 97% which shows a 2-level predictor learns the correlations well (Niagara has no branch prediction). Columns 7 and 8 show IPC which averages close to 1 on the 3-issue out-of-order Clovertown, and about 0.4 on the in-order Niagara, showing limited ILP.

Table 1 shows our coarse-grained analysis of computation. Six functions (lines 11 through 16 in Table 1) contribute more than 80% of the total computation time. Most of the computation time is spent in the top three functions which take a packet of rays and traverse the kd-tree data structure. As needed, they call other functions to lazily build the tree during traversal. When rays do intersect a leaf node, `IntersectRayKDLeaf` is called to traverse a smaller spatial data structure for a tessellated surface patch. As needed, it, in turn, calls another function to lazily create the geometry. Actual ray/triangle intersection is performed by the fifth and sixth most common functions which prepare the pair of triangles to interpolate from discrete levels of detail and perform the actual intersection test on the interpolated triangle.

Observation 6: There is limited fine-grained instruction-

Scene	Instruction Mix(%)					IPC		Speedup	
	ALU	FP	LD	ST	BR	N	C	OOO	SSE
courtyard	42	32	14	8	2	0.4	1.01	3.4	3.5
fairyforest	46	26	13	8	4	0.3	1.03	3.4	3.1
forest	53	18	15	5	7	0.4	1.00	2.6	4.2
juarez	49	18	12	12	6	0.4	1.05	3.3	3.7
saloon	48	22	13	9	6	0.3	1.04	3.7	3.4

Table 2. Razor computation characteristics. Instruction mix data shown is for Niagara SPARC code with SIMD emulation. Pin-based x86 instruction mix is similar.

Core type	# Cores	Rendering time(sec)	# Cores	Rendering time(sec)
	256KB cache per core		Fixed 2MB cache total	
OOO*	13	3.7 - 1.5	44	7 - 0.5
Inorder+	109	33 - 2.7	167	22 - 1.8

Table 3. Simple optimistic scaling of existing designs. Performance range across the different scenes. * Core2-like core. + Niagara-like core.

level parallelism and significant amount of irregular graph traversal.

Machine-specific optimizations: The last two columns in Table 2 compare performance across our two prototype systems, which shows Razor's sensitivity to out-of-order processing and ISA specialization. To normalize for technology, Clovertown system performance was linearly scaled down by the frequency difference to Niagara. Speedup in this *scaled* rendering time is compared to Niagara. Adding out-of-order processing and increased issue width alone provides an approximate 3X performance improvement, shown by the OOO column. Adding SIMD specialization with SSE3 instructions and machine specific optimizations (compilation with `gcc -O4 -msse3`) provides an additional 3X to 4.2X improvement, shown in the last column.

Observation 7: Application specific tuning can provide significant performance improvements for Razor.

This workload characterization of Razor shows that it is highly parallel and is thus suited for a multicore parallel architecture. But it is not trivially data parallel, and while its memory accesses are quite irregular, they have good locality.

4. Architecture

Our goal is to build a future system targeted at high quality scenes at 1024x1024 resolution at real time rates implying rendering times of 0.04 seconds per frame. On average, performance on our 8-core system ranges between 17.5 to 4 seconds per frame for high quality scenes and between 1.1 to 0.5 seconds for lower quality scenes. Compared to this system, we need another 100-fold to 500-fold increase in performance. While Razor is heavily optimized, we believe further algorithmic tuning is possible and expect a 10-fold increase in its performance and thus our target is 0.4 seconds per frame using the current software. Thus the architecture must provide 10- to 50-fold better performance than our 8-core prototype system. We pick a design point fives years in the future and target 22nm technology, a frequency of 4GHz which should be feasible with relatively modest timing and pipelining optimizations, and die area of $240mm^2$.

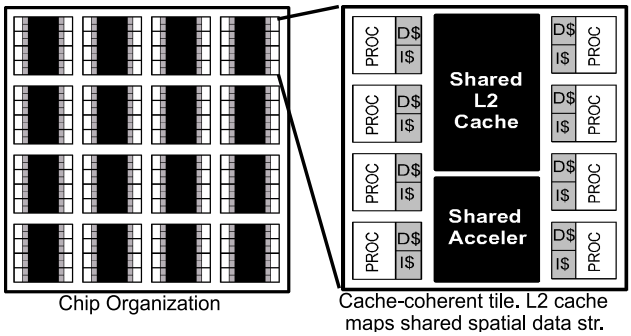


Figure 4. Copernicus hardware organization.

Simply scaling existing systems has three main problems: a) area constraints limit the performance that can be achieved, b) the on-chip memory system and interconnect design becomes overly complex, and c) general-purpose solutions do not exploit several opportunities for application-specific tuning. Table 3 summarizes the area and performance that can be sustained by simply scaling and integrating more cores on chip. The first method assumes the cache capacity is scaled such that there is always 256KB of L2 cache per core and the second method provides a total of 2MB for all the cores. Examining the OOO (out-of-order) cores, they are simply too large to provide the kind of parallel speedup required. The in-order Niagara-like cores, while area efficient, do not have support for fine-grained data-level parallelism and their baseline performance is too low. Thus neither technique provides the performance required. What is ideally required is Core2-like performance with the area of an in-order processor. Second, existing multicore designs of shared L2 caches with global coherence don't scale to hundreds of cores. Finally, the cumulative L2 miss-rates of all the cores places a large burden on the memory system, and the fixed 2MB L2 cache will face severe congestion and require several ports. In this section, we describe the hardware architecture of our Copernicus system which exploits application behavior and fuses existing technologies and develops new techniques.

4.1. High-level organization

We propose a 2-level multicore organization with a set of cores and a shared L2 cache forming a tile. The full chip consists of multiple tiles (caches private to tile) and I/O controllers. Figure 4 shows this organization and we specifically investigate a 16-tile, 8 cores per tile configuration.

Figure 1 shows the software mapping to the architecture. The 2D screen-space is partitioned and into at least as many "blocks" as there are tiles, and coarse grained parallelism is

achieved by providing a "block" to each tile. These "blocks" are further broken into "chunks", the granularity at which Razor operates via its chunk work queue. Chunks are sized to capture significant memory-access locality, made large enough to amortize the cost of communication with the work queue, and small enough to not cause load-imbalance. Within a "chunk", each core ray-traces packets of up to 16 light rays together exploiting fine-grained parallelism.

4.2. Core organization

Each core in a tile resembles a conventional programmable core but with specializations for processing-intensive workloads such as graphics rendering, by matching the memory intensity and computation requirements. Since it is not possible to keep the entire working set in the L1 cache, techniques to hide this latency are required. For this workload, out-of-order processing is not area or power efficient, and since there is abundant parallelism, we use simultaneous multi-threading to hide memory latency. Unlike GPUs, which require hundreds of threads per core to hide main memory latency, we expect good L2 cache behavior and thus require only modest amount of SMT - between 2 and 8 threads per core. Razor is a packet ray-tracer and exhibits fine-grained parallelism in ray processing. In fact this is fine-grained data-level parallelism because the same graph traversal and surface-ray intersection operations are computed for every ray in a packet. To support this efficiently, each core also includes a 4-wide SIMD unit. Our initial design explored in this paper uses the Intel SSE intrinsics for our SIMD unit as the software was already tuned for it.

In addition, each tile includes an "accelerator" that is integrated into the execution pipeline, although this accelerator is not currently used by our software. Long latency and highly specialized tasks could be routed to this unit by all the cores in a tile. We expect moderately frequent computation like square root, trigonometric equations, and light attenuation to be implemented in the accelerator. Having such a shared per-tile accelerator can provide significant area and power efficiency, to be explored in future work. In this study, we only assume an SSE-like unit in every core.

4.3. On-chip memory and Memory system

Each tile includes a shared L2 cache that is connected through a cross-bar to the different cores. The different tiles are connected through a mesh-network that eventually routes requests to the integrated memory controllers. Our characterization work suggests that in the worst case, a single frame touches a total of 122MB to approximately 600MB worth of cache lines (Table 1, row 4). Since our system rebuilds its spatial data structures every frame, we anticipate a worst case memory bandwidth of 600MB * 25 frames/second = 15GB/second. Detailed memory system tuning is beyond the scope of this work and in this study we use a conservative analytical model to estimate memory system contention.

Our L1-L2 cache is designed for simplicity and follows a simple cache coherence protocol. The L1 caches are write through and the L2 caches maintain a copy of the L1 tags and keep track of which core in a tile has a copy of any given L1 line. A significant simplification in our design which allows us to scale to the order of 128 cores is that the hardware provides no coherence guarantees across tiles. This abstraction works for this application but how well this extends to other applications is an open question. Given technology trends of wire-delays and storage, this tradeoff of redundancy over synchronization is a unique design point that has not been explored in the past.

5. Results

This section evaluates the Copernicus system. We use the five benchmark scenes described in Table 1 and a full system simulator and analytical models for performance estimates. We first present baseline performance that is achievable on a single-core, and then study performance improvements through multi-threading and parallelism from multiple cores on a tile. Finally, we present full chip results using the analytical model.

5.1. Methodology

Simulator: We customized Multifacet GEMS to model a single tile with in-order issue and varying level of multi-threading support. The simulator is configured with a 32KB, 2-way set associative L1 data and instruction cache. The shared L2 cache is set to be 2MB, and 4-way set associative. We use the default GEMS functional unit latencies and a 2-level branch prediction scheme with a total of 100Kbits of target and history. Thread-scheduling logic is strict round-robin between the threads that are not waiting on a L1 cache miss. Razor is compiled for a sparc-sun-solaris2.10 target with our C-only SSE intrinsics library. The implementation triggers a trap in GEMS, and we simulate 4-wide SIMD execution of these instructions, ignoring the functional instruction stream.
Datasets: Rendering a full frame of a complete scene (or multiple scenes) is truly representative of performance but the simulation overhead for this is too high - a single frame is about 50 billion instructions. Thus it is necessary to simulate a subset of a frame, but taking care to account for the significant amount of intra-scene heterogeneity. For example, background regions require little computation, whereas regions along a shadow boundary require lots of computation. To account for this heterogeneity, we simulate four randomly selected regions for all the benchmarks, and for every region we simulated the smaller of 200 million instructions or 128 primary rays (and all of their accompanying secondary rays) per thread. The performance we report is the average across these regions.

Metrics: It is easier to compare rendering systems and understand tradeoffs by measuring total rays rendered per second rather than frames/second. In the remainder of this paper we use total rays/sec to report performance and IPC to report processor utilization. Razor automatically generates secondary rays as the primary rays traverse the scene and all the scenes used in this study require 40 million total rays to render the full scene (1024x1024 resolution, with 8 primary rays per pixel and 4 secondary rays per primary ray for our scenes). A real-time rate of 25 frames-per-second translates to 1,000 million total rays/sec. Other contemporary systems like Cell's iRT sustain roughly 46 million total rays/second on static game-like scenes with shading [1]. Note that we target significantly higher image quality. We assume a ten-fold improvement in Razor's algorithms is possible, and hence our goal with this architecture is to sustain 100 million total rays/sec on the current Razor software.

Limitations: In this work, we abstract away the details of the operating system and thread scheduling. Our system simulation relies on the operating system's scheduling heuristics. We did not observe large load imbalances. The number of threads created is under application control and we set it to match number of threads the hardware supports.

5.2. Analytical model

Microarchitecture simulation of the full architecture is infeasible. So, we developed a new customized analytical model that takes fundamental input parameters which characterize the system and workload, and projects performance - at scale - for a variety of system design alternatives.

The model accounts for all non-ideal parallel scaling of the workload on the hardware. More specifically, the model captures the longer communication latencies to main memory and three types of contention: (1) the intra-core contention among threads for the processor core, (2) the intra-tile contention for the shared L2 caches, and (3) the intra-chip contention for main memory. Since each tile has its own k-d tree, synchronization delays for nodes in the tree, which are negligible in the single-tile case, do not increase with more tile.

The model provides insights into the impact of pipeline stalls within a thread, processor stalls in the context of multiple threads, and contention for memory resources. Accurate modeling of the thread contention for core processor cycles requires modeling the impacts of instruction dependences as well as multi-issue within the core pipeline. This, in turn, required new fundamental model input parameters and new model equations. These new model customizations are applicable to evaluating other applications on other future multi-threaded multicore architectures, and to our knowledge have not previously been developed.

Since our processors are in-order, the main parameters that dictate the processor's execution are the L1 and L2 cache misses and the register dependences between instructions. The dependences are measured in terms of the distribution of the distance from each instruction to its first dependent instruction. We outline the details of the model in three stages.

Pipeline stalls: In the first stage, the fundamental workload instruction mix parameters given in Table 2 are used together with the functional unit latencies to estimate, for each instruction type i, the average time that the next instruction will stall due to either a dependence or a shorter latency. We let f_i denote the fraction of instructions that are of type i, L_i denote the instruction pipeline length for instruction type i, and d_1 denote the fraction of instructions with distance to first dependent instruction that are equal to one. Letting t_i denote one (for the cycle that type i begins execution) plus the average stall time for the next instruction, we capture the first-order pipeline stall effects for this workload as follows:

$$t_i = 1 + \sum_{j:L_j<L_i, j!=branch} f_j(L_i - L_j)$$
$$+(\sum_{L_j>=L_i} f_j + f_{branch}) \times d_1 \times (L_i - 1)$$

Multi-threading: In the second stage, we use the above equation to first compute t, the sum of t_i weighted by f_i, and then to compute the distribution of the number of instructions that begin executing in each cycle. This latter quantity is a function of the number of threads that are being interleaved (k), and the instruction issue width. For k threads on a single issue core, the fraction of cycles that an instruction is issued can be modeled by, $1 - \left(1 - \frac{1}{t}\right)^k$. In this case, $r(k)$, the ray tracing rate with k threads as compared with one thread is the ratio of this quantity to $1/t$, since $1/t$ is the fraction of cycles that an instruction is issued when a single thread executes on a single-issue core. It is similarly straightforward to develop the execution rates for a 2-issue core.

Memory system: In the final stage of model development, we incorporate the impact of contention for L2 and main memory banks. Here we use well-established queuing network approximate mean value analysis (AMVA) techniques [15]. Each cache and main memory bank is represented by simple single-server queues with deterministic service times, routing of cache misses is represented by visit counts to the applicable L2 cache and main memory banks, interconnection network latencies are represented by delay centers, and each core is represented by a variable-rate queue to capture the execution rate as a function of the number of threads that are executing (and not waiting on a cache miss). Equations for such queuing systems are available for example in [15]. Such equations have proven to be accurate in other parallel system architectures with a single-thread per core [32]. A key contribution of this paper is the validation of the extensions to estimate pipeline stalls and multi-threading.

Using this fairly simple system of equations, we explore the

Scene	1-Thread			2-Threads			4-Threads		
	IPC		Perf.	IPC		Perf.	IPC		Perf.
	S	M		S	M		S	M	
	Single core, 1-issue processor								
courtyard	0.45	0.46	280	0.82	0.71	515	0.97	0.92	605
fairyforest	0.39	0.43	480	0.73	0.69	900	0.97	0.91	1200
forest	0.43	0.52	125	0.86	0.78	255	0.97	0.95	285
juarez	0.34	0.38	200	0.69	0.70	400	0.97	0.93	565
saloon	0.38	0.43	360	0.73	0.70	590	0.97	0.92	915
	Single core, 2-issue processor								
courtyard	0.7		440	1.1	0.84	690	1.7	1.5	1065
fairyforest	0.59		730	0.94	0.72	1160	1.7	1.2	2100
forest	0.64		190	1.2	1.03	355	1.9	1.6	560
juarez	0.48		280	0.9	0.80	525	1.7	1.4	990
saloon	0.56		530	0.98	0.80	925	1.7	1.4	1605
	One Tile - 8 cores per tile, 1-issue processors								
courtyard	3.7	3.7	2280	7.0	5.7	4375	7.8	7.4	4875
fairyforest	3.6	3.4	4445	7.2	5.5	8890	7.8	7.3	9630
forest	3.8	4.1	1120	7.2	6.3	2125	7.9	7.6	2330
juarez	3.7	3.5	2150	7.4	5.6	4300	7.8	7.4	4535
saloon	3.6	3.6	3395	7.1	5.6	6700	7.8	7.3	7370
	One Tile - 8 cores per tile, 2-issue processors								
courtyard	5.0		3125	8.8	6.7	5500	14.8	11.8	9250
fairyforest	4.9		6050	8.8	6.1	10865	14.6	10.8	18025
forest	5.3		1565	8.9	8.3	2625	14.6	13.3	4305
juarez	4.8		2790	9.0	6.7	5235	14.6	14.8	8490
saloon	5.0		4715	8.2	6.4	7745	14.3	11.5	13510

Table 4. Single tile model validation. Column S: simulator results. Column M: analytical model results. Performance measured in 1000s of rays/sec from simulator.

design space of numer of tiles, cores, threads-per-core, issue-width, and scene-specific cache miss rates, and provisioning of how many memory banks. Other model-derived insights include:

Model Insight 1: Stalls due to dependences have small impact on performance.

Model Insight 2: Stalls due to L1 cache misses have a large impact on performance.

Model Insight 3: Significant speedup is possible with multi-threading and two-issue cores, but these speedups come at the cost of significant demands for memory bandwidth.

5.3. Performance results

We first examine single-tile performance from simulation and the model to validate our model. We then examine the full system performance and total chip area.

Single-tile and model validation: Table 4 shows the performance across the different processor configurations we study, which all assume a 4GHz clock frequency. We specifically examine the sensitivity to multi-threading and issue width. For each SMT configuration, there are three columns: IPC from the simulator, IPC from the model, and performance (total rays per second in 1000s). Considering the single core, 1-thread configuration, we can see that a single thread is able to utilize less than 50% of the processor for almost all the scenes, and with 2 threads close to 80% is utilized (IPC is 0.69 to 0.83), and reaches almost 100% at 4 threads (IPC is 0.97). Our analytical model predicts performance accurately

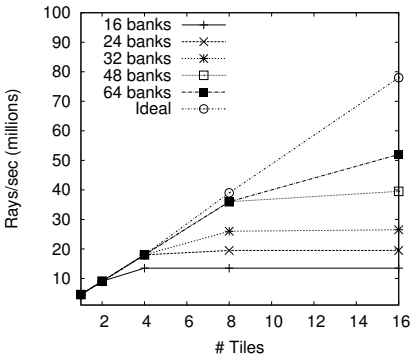


Figure 5. Performance sensitivity to memory bandwidth. 1-issue, 4-way SMT core. [Courtyard scene].

(IPCs in the M column) and is off by less than 2%. The second set of rows, show performance for a 2-issue processor. The 1-thread and 2-thread configurations perform moderately, and we see large benefits when executing 4 threads with IPCs ranging for 1.7 to 1.9. The model errors are slightly higher.

The last two sets of results are for a full tile consisting of 8 cores and we show aggregated IPC from all cores. The single issue cores are more efficient and come close to executing at 100% utilization with 4 threads with IPCs reaching 8. The deviation of the analytical model from simulation is now higher. Overall though, it is consistently more conservative than the simulator because of how it models L2-bank contention.

The IPC by itself is a meaningless number when determining performance for the scenes, as each scene requires different instructions and amount of work. Instead, total rays per second is a better measure. Our target for real-time rates is 100 million total rays/second using all the tiles. Table 4 shows that a single tile with single-issue and four threads per core can sustain between 2.3 million and 9.6 million rays per second.

Full system design exploration: To obtain full-chip performance, we used our full system analytical model, which models memory and on-chip network contention. To complete the chip design, we need to determine memory system configuration and interconnection network.

Memory: The model can determine memory bandwidth required and the number of memory banks so that enough overlapping requests can be serviced. We used the model to analyze our different workloads by varying the number of tiles and number of memory banks. Figure 5 shows this analysis for one benchmark `courtyard` for a single-issue 4-way multithreaded core, and the behavior was largely similar for all benchmarks. With 64 banks, the performance is within 50% of ideal (no contention). This gives us a realistic system of 4 DIMMs with 16 banks each.

Interconnection network: Most of the contention for this workload is at the memory system and not the mesh network on chip, since assuming a shared bus to model worst case network contention, results in the memory system being the

Core type	Performance (millions rays/sec)				#Tiles	#Cores
	Ideal		Realistic			
	Ave.	Range	Ave.	Range		
1-issue, no SMT	37	16-62	42	19-66	16	128
1-issue, 4 threads	**82**	**33-138**	**43**	**13-74**	**16**	**115**
2-issue, no SMT	34	15-57	26	12-41	10	78
2-issue, 4 threads	82	36-137	41	13-70	8	65

Table 5. Full system performance and area for all scenes. We recommend 1-issue, 4 threads as the best configuration.

bottleneck and not the network.

Full system performance: With the number of memory banks specified, our full system model can make performance projections for different core configurations and scenes which are shown in Table 5 as an average and range across our five benchmark scenes. Columns two and three show "ideal" performance achieved by naively scaling the performance for one tile to 16 tiles (or as many that will fit), ignoring contention for shared resources. Columns four and five show realistic performance with the 4-DIMM memory. Our conservative memory model projects that we can sustain between 13 and 74 million rays/second, and the ideal model suggests 33 to 138 million rays/second.

Full system area analysis: Using die-photos of other processors we built an area model for the tiles and the full chip. For overheads of multi-threading, we use methodology from academic and industry papers on area scaling [19]. The area for our baseline 1-issue core is derived from a Niagara2 core scaled to 22nm, and we assume a 12% area overhead for multi-threading for 2-threads, and additional 5% per thread. In our area analysis, the additional area of the 4-wide FPU compared to Niagara2's FPU is ignored. We assume an 110% overhead when going from 1-issue to 2-issue, since in this model we double the number of load/store ports also and hence the L1 cache sizes. The cache size is fixed at 2MB for all designs and this includes the cross-bar area between the cores and the cache. Columns 6 and 7 in Table 5 show the number of cores of each type that will fit on a $240mm^2$ chip. With single-issue cores, we can easily fit 16 tiles, and about half with two issue cores.

Summary: Overall, going to dual issue does not provide significantly higher performance and single-issue, 4-way multithreaded cores seem ideal. Single-issue cores without SMT seem to match the 4-way SMT cores at the chip-level, because of insufficient memory bandwidth to feed the threads. At the chip-level, memory bandwidth is the primary bottleneck. While our final performance results do not outright exceed the required 100 million rays/second for every scene, the flexibility and potential for further architectural optimizations show this is a viable system.

6. Related work

Recently, a few application-driven architectures have been proposed. Yeh et al. have proposed ParallAX, an architecture specialized for real-time physics processing [42]. Hughes et al. recently proposed a 64-core CMP for animation and visual effects [10]. Kumar et al. have proposed an architecture called Carbon that can support fine-grained parallelism for large scale chip-multiprocessors [14]. Yang et al., describe a scalable streaming processor targeted at scientific applications [41]. Clark et al. have proposed a technique called liquid SIMD that can abstract away the details of the data-parallel hardware [4]. Sankaralingam et al. developed a methodical approach for characterizing data parallel applications and proposed a set of universal mechanisms for data parallelism [29]. All these designs follow the flow of workload characterization of an *existing application* driving the design of an architecture. Our work is different in two respects. First, we co-design *future* challenge applications and the architecture to meet the application performance requirements. Second, we provide a more general purpose architecture and new quantitative tools to support the design process. Embedded systems use similar hardware/software co-design but for building specialized processors and in much smaller scale. Architecture-specific analytical models have been applied for processor pipelines to analyze performance and power [22], [37], [13], [34], [43], [8], [27]. These models have also been used for design space exploration and rapid simulation [12], [11], [16], [3], [6], [24].

7. Discussion and Future work

Building efficient systems requires that the software and hardware be designed for each other. Deviating both hardware and software architectures from existing designs poses a particularly challenging co-design problem. The focus of this paper has been the co-design of the software, architecture, and evaluation strategy for one such system: a ray-tracing based real-time graphics platform that is fundamentally different from today's Z-buffer based graphics systems.

Our software component (Razor) was designed to allow graphics and scene behavior to be exploited by hardware via fine-grained parallelism, locality in graph traversal, good behavior of secondary rays, and slow growth in memory by virtue of kd-tree duplication. To solve the application/architecture "chicken-and-egg" problem, we implemented Razor on existing hardware, in our case SSE-accelerated x86 cores and built a prototype system using available technology with effectively eight cores. This prototype allowed us to perform detailed application characterization on real scenes, something impossible on simulation and meaningless without an optimized application.

Closing the development loop with design, evaluation and analysis of software's behavior on our proposed hardware architecture was accomplished via a novel analytical model that provided intuition both for the architecture and the application. For example, an important application design

question was to decide whether to duplicate or share the kd-tree data structure. Our 8-core prototype hardware system does not scale and our simulator for proposed hardware is too slow to run the full application to completion. Only the model could reveal that duplication overhead is manageable, thus relaxing coherence requirements for the architecture. To develop the model, we combined data gathered on the optimized prototype using tools like Pin and Valgrind with simulation-based measurements of cache behavior and instruction frequency. With this co-designed approach, we have shown the raytracing-based *Copernican* universe of application and general visibility-centric 3D graphics is feasible. However, this work represents only a first cut at such a system design; there is more to explore in the application, architecture, and evaluation details.

Application: Razor is the first implementation of an aggressive software design incorporating many new ideas, some of which have worked better than others. With further iterative design, algorithm development, and performance tuning, we believe a ten-fold performance improvement in the software is possible. Non-rendering tasks in a game environment and more generally irregular applications map well to the architecture and require further exploration.

Architecture: There is potential for further architectural enhancements. First, the length of basic blocks is quite large and hence data-flow ISAs and/or greater SIMD width can provide higher efficiency. Second, ISA specialization, beyond SIMD specialization targeted at shading and texture computations could provide significant performance improvements. Improving memory system will be most effective, and 3D integrated DRAM could significantly increase performance and reduce system power [17]. Physically scaling the architecture by varying the number of tiles, frequency, and voltage scaling to meet power and area budgets provides a rich design space to be explored.

Evaluation: Our analytical model enables accurate performance projections and can even be used for sensitivity studies. In addition, it can be extended to accommodate other Copernican architectures, like Intel Larrabee [30].

Comparison to GPUs and Beyond Ray-tracing: The processor organization in Copernicus is fundamentally different from conventional GPUs, which provide a primitive memory system abstraction while deferring scene geometry management to the CPU. Architecturally, the hardware Z-buffer is replaced with a flexible memory system and software spatial data structure for visibility test. This support enables scene management and rendering in one single computational substrate. We believe GPUs are likely to evolve to such a model over time, potentially with a different implementation. For example, secondary rays could be hybridized with Z-buffer rendering. Our system is a particular point in the architecture design space that can support ray tracing as one of potentially several workloads.

8. Conclusions

Modern rendering systems live in a *Ptolemic* Z-buffer universe that is beginning to pose several problems in providing significant visual quality improvements. We show that a *Copernican* universe centered around applications and sophisticated visibility algorithms with ray-tracing is possible and the architecture and application challenges can be addressed through full system co-design. In this paper, we describe our system, called Copernicus, which includes several co-designed hardware and software innovations. Razor, the software component of Copernicus, is a highly parallel, multi-granular, locality-aware ray tracer. The hardware architecture is a large-scale tiled multicore processor with private L2 caches, fine-grained ISA specialization tuned to the workload, multi-threading for hiding memory access latency, and limited (cluster-local) cache coherence. This organization represents a unique design point that trades off data redundancy and recomputation over synchronization, thus easily scaling to hundreds of cores.

The methodology used for this work is of interest in its own right. We developed a novel evaluation methodology that combines software implementation and analysis on current hardware, architecture simulation of proposed hardware, and analytical performance modeling for the full hardware-software platform. Our results show that if projected improvements in software algorithms are obtained, we can sustain real-time raytracing on a future $240mm^2$ chip at 22nm technology. The mechanisms and the architecture are not strictly limited to ray-tracing, as future systems that must execute irregular applications on large scale single-chip parallel processors are likely to have similar requirements.

Acknowledgment

We thank Paul Gratz, Boris Grot, Simha Sethumadhavan, the Vertical group, and the anonymous reviewers for comments, the Wisconsin Condor project and UW CSL for their assistance, and the Real-Time Graphics and Parallel Systems Group for benchmark scenes and for their prior work on Razor. Many thanks to Mark Hill for several valuable suggestions. Support for this research was provided by NSF CAREER award #0546236 and by Intel Corporation.

References

[1] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich, "Ray Tracing on the CELL Processor," in *Interactive Ray Tracing*, 2006, pp. 15–23.

[2] J. Bigler, A. Stephens, and S. Parker, "Design for parallel interactive ray tracing systems," in *Interactive Ray Tracing*, 2006, pp. 187–196.

[3] D. Brooks, P. Bose, V. Srinivasan, M. K. Gschwind, P. G. Emma, and M. G. Rosenfield, "New methodology for early-stage, microarchitecture-level power-performance analysis of microprocessors," *IBM J. Res. Dev.*, vol. 47, no. 5-6, pp. 653–670, 2003.

[4] N. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner, "Liquid SIMD: Abstracting SIMD Hardware using Lightweight Dynamic Mapping," in *HPCA '07*, pp. 216–227.

[5] P. Djeu, W. Hunt, R. Wang, I. Elhassan, G. Stoll, and W. R. Mark, "Razor: An architecture for dynamic multiresolution ray tracing," *Conditionally accepted to ACM Transactions on Graphics (available as UT Austin CS Tech Report TR-07-52)*.

[6] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. D. Bosschere, "Statistical simulation: Adding efficiency to the computer designer's toolbox," *IEEE Micro*, vol. 23, no. 5, pp. 26–38, 2003.

[7] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *SIGPLAN '82: Symposium on Compiler Construction*, pp. 120–126.

[8] A. Hartstein and T. R. Puzak, "The optimum pipeline depth for a microprocessor," in *ISCA '02*, pp. 7–13.

[9] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree gpu raytracing," in *I3D '07*, pp. 167–174.

[10] C. J. Hughes, R. Grzeszczuk, E. Sifakis, D. Kim, S. Kumar, A. P. Selle, J. Chhugani, M. Holliman, and Y.-K. Chen, "Physical simulation for animation and visual effects: parallelization and characterization for chip multiprocessors," in *ISCA '07*, pp. 220–231.

[11] E. Ipek, S. A. McKee, K. Singh, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficient architectural design space exploration via predictive modeling," *ACM Trans. Archit. Code Optim.*, vol. 4, no. 4, pp. 1–34, 2008.

[12] T. Karkhanis and J. E. Smith, "Automated design of application specific superscalar processors: an analytical approach," in *ISCA '07*, pp. 402–411.

[13] ——, "A first-order superscalar processor model," in *ISCA '04*, pp. 338–349.

[14] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," in *ISCA '07*, pp. 162–173.

[15] E. Lazowska, J. Zahorjan, G. Graham, , and K. Sevcik, *Quantitative System Performance*. Prentice-Hall, 1984.

[16] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *ASPLOS-XII*, pp. 185–194.

[17] G. H. Loh, "3D-Stacked Memory Architectures for Multi-core Processors," in *ISCA '08*, pp. 453–464.

[18] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *Computer Architecture News (CAN)*, 2005.

[19] H. M. Mathis, A. E. Mericas, J. D. McCalpin, R. J. Eickemeyer, and S. R. Kunkel, "Characterization of simultaneous multithreading (smt) efficiency in power5," *IBM Journal of Research and Development*, vol. 49, no. 4-5, pp. 555–564, 2005.

[20] S. Moore, D. Terpstra, K. London, P. Mucci, P. Teller, L. Salayandia, A. Bayona, and M. Nieto, "PAPI Deployment, Evaluation, and Extensions," in *DOD-UGC '03*, pp. 349–353.

[21] N. Nethercote, R. Walsh, and J. Fitzhardinge, "Building Workload Characterization Tools with Valgrind," in *IISWC '06*, p. 2.

[22] D. B. Noonburg and J. P. Shen, "Theoretical modeling of superscalar processor performance," in *MICRO '94*, pp. 52–62.

[23] S. Parker, W. Martin, P.-P. J. Sloan, P. Shirley, B. Smits, and C. Hansen, "Interactive ray tracing," in *SIGGRAPH '05 Courses*.

[24] D. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. August, and D. Connors, "Exploiting parallelism and structure to accelerate the simulation of chip multi-processors," in *HPCA '06*, pp. 29–40.

[25] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[26] "Pin: http://rogue.colorado.edu/wikipin/index.php/downloads."

[27] T. R. Puzak, A. Hartstein, V. Srinivasan, P. G. Emma, and A. Nadas, "Pipeline spectroscopy," in *SIGMETRICS '07*, pp. 351–352.

[28] E. Reinhard, B. E. Smits, and C. Hansen, "Dynamic acceleration structures for interactive ray tracing," in *Eurographics Workshop on Rendering Techniques*, 2000, pp. 299–306.

[29] K. Sankaralingam, S. W. Keckler, W. R. Mark, and D. Burger, "Universal Mechanisms for Data-Parallel Architectures," in *MICRO '03*, pp. 303–314.

[30] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," in *SIGGRAPH '08*, 2008, pp. 1–15.

[31] B. Smits, "Efficiency issues for ray tracing," *J. Graph. Tools*, vol. 3, no. 2, pp. 1–14, 1998.

[32] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood, ""analytic evaluation of sharedmemory parallel systems with ilp processors"," in *Proc. 25th Int'l. Symp. on Computer Architecture (ISCA '98)*, 1998, pp. 380–391.

[33] J. Spjut, S. Boulos, D. Kopta, E. Brunvand, and S. Kellis, "Trax: A multi-threaded architecture for real-time ray tracing," in *SASP '08*, pp. 108–114.

[34] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. V. Zyuban, P. N. Strenski, and P. G. Emma, "Optimizing pipelines for power and performance," in *MICRO '02*, pp. 333–344.

[35] E. B. Sven Woop and P. Slusallek, "Estimating performance of a ray-tracing asic design," in *Interactive Ray Tracing*, September 2006, pp. 7–14.

[36] J. S. Sven Woop and P. Slusallek, "RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing," in *SIGGRAPH '05*, pp. 434 – 444.

[37] T. Taha and D. Wills, "An instruction throughput model of superscalar processors," in *Workshop on Rapid Systems Prototyping '03*, pp. 156–163.

[38] I. Wald, C. Benthin, and P. Slusallek, "Distributed interactive ray tracing of dynamic scenes," in *PVG '03*, p. 11.

[39] I. Wald, T. J. Purcell, J. Schmittler, C. Benthin, and P. Slusallek, "Realtime Ray Tracing and its use for Interactive Global Illumination," in *Eurographics State of the Art Reports*, 2003.

[40] S. Woop, G. Marmitt, and P. Slusallek, "B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes," in *Proceedings of Graphics Hardware*, 2006, pp. 67–77.

[41] X. Yang, X. Yan, Z. Xing, Y. Deng, J. Jiang, and Y. Zhang, "A 64-bit stream processor architecture for scientific applications," in *ISCA '07*, pp. 210–219.

[42] T. Y. Yeh, P. Faloutsos, S. J. Patel, and G. Reinman, "Parallax: an architecture for real-time physics," in *ISCA '07*, pp. 232–243.

[43] V. V. Zyuban, D. Brooks, V. Srinivasan, M. Gschwind, P. Bose, P. N. Strenski, and P. G. Emma, "Integrated analysis of power and performance for pipelined microprocessors," *IEEE Trans. Computers*, vol. 53, no. 8, pp. 1004–1016, 2004.