Graphics Hardware (2002), pp. 1–11
Thomas Ertl, Wolfgang Heidrich, and Michael Doggett (Editors)

SaarCOR — A Hardware Architecture for Ray Tracing

Jörg Schmittler, Ingo Wald, Philipp Slusallek

Computer Graphics Group, Saarland University, Germany

Abstract

The ray tracing algorithm is well-known for its ability to generate high-quality images and its flexibility to support advanced rendering and lighting effects. Interactive ray tracing has been shown to work well on clusters of PCs and supercomputers but direct hardware support for ray tracing has been difficult to implement.
In this paper, we present a new, scalable, modular, and highly efficient hardware architecture for real-time ray tracing. It achieves high performance with extremely low memory bandwidth requirements by efficiently tracing bundles of rays. The architecture is easily configurable to support a variety of workloads. For OpenGL-like scenes our architecture offers performance comparable to state-of-the-art rasterization chips. In addition, it supports all the usual ray tracing features including exact shadows, reflections, and refraction and is capable of efficiently handling complex scenes with millions of triangles. The architecture and its performance in different configurations is analyzed based on cycle-accurate simulations.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism B.5.1 [Hardware Design]: Register-Transfer-Level Implementation

1. Introduction

Currently, interactive computer graphics is almost exclusively available through graphics hardware based on the rasterization algorithm. Over the last few years hardware integration allowed this approach to offer high performance 3D graphics even on low cost PCs.

However, rasterization hardware is severely limited by memory bandwidth due to frequent overwrites of pixels in the framebuffer and its heavy use of Z and stencil buffers. Current graphics chips require the latest memory technology and high memory clock rates to reach their performance.

Additional limitations result from the strict pipeline structure of the rasterization hardware. The implementation of advanced rendering effects requires complex, non-intuitive, and non-trivial use of textures, multiple rendering passes, and complex pipeline programming. Even then, most effects can only be approximated because their complex sampling pattern cannot be supported efficiently, even for simple cases such as reflections.

Furthermore, rasterization performance decreases linearly with the number of triangles in a scene. This requires complex algorithms, such as geometry simplification or occlusion culling, to be implemented by applications in order to reduce the number triangles to be submitted to the hardware.

Ray tracing does not suffer from these problems. It scales logarithmically in the number of triangles using a simple hierarchical acceleration structure and occlusion culling comes for free [30]. The requirements for shading and frame buffer access are low as every ray gets shaded exactly once and no Z buffer tests are required to resolve visibility. Ray tracing can also be parallelized easily as each ray is essentially independent of all other rays.

Because ray tracing is not limited to a strict pipeline structure, shading can be separated from visibility computations and can be executed asynchronously. This allows the direct use of sophisticated shaders (like RenderMan [8]) that result in physically correct images — providing correct shadows, multiple reflections on arbitrary surfaces, complex reflectance functions, and correct visual effects, such as transmission and refraction.

Attempts to directly implement ray tracing in hardware has caused many problems due to its use of recursion, the high requirements for floating point computation and its irregular and high bandwidth memory access patterns. This

has made ray tracing rather unattractive for a hardware implementation.

The ray tracing architecture presented in this paper is based on the observation that significant coherence between rays does exist and can be exploited [30] by rearranging the core ray tracing algorithm. While this has worked well already for a software implementation it is even more attractive for hardware.

We show that for OpenGL-like scenes ray tracing hardware offers performance comparable to state-of-the-art rasterization hardware with significantly lower bandwidth requirements. In addition, ray tracing hardware provides much higher image quality and handles significantly larger scenes than current rasterization hardware.

In this paper, we concentrate on the ray traversal and intersection part of a ray tracing architecture and only briefly consider shading with a fixed Phong-like reflection model and a single bilinearly filtered texture. In a related publication [28] we show that dynamic changes of the scene can effeciently be supported even for a ray tracer. With OpenRT [29] we have also proposed an API for ray tracing. This API is based on OpenGL but has been extended to better support the requirements of ray tracing based renderers.

Using a configuration of our architecture that is comparable to the complexity[†] of current rasterization hardware, we achieve more than 200 frames per second (fps) for typical OpenGL-like scenes, rendered at 1024x768 pixels. The standard configuration of our architecture, which has roughly half the cost of current rasterization hardware, still renders highly complex scenes with millions of triangles, and high-quality shading including multiple reflections and shadows from several light sources at more than 25 fps.

2. Previous Work

Interactive 3D graphics today is entirely dominated by rasterization hardware[1]. Today's consumer graphic cards require significant compute power and memory bandwidth to achieve their level of performance. For example Nvidia's GeForce 3[10] offers 76 GFlops at a clock rate of 200 MHz and has a 256 bit wide memory interface running at 230 MHz. These results require at least 380 parallel floating point units and offer a memory bandwidth of 7.2 GB/s.

A major issue with most rasterization approaches is scalability. One exception is the Pomegranate architecture [2]. However this architecture requires high bandwidth connections as well as complex routing schemes between the functional units.

In contrast to rasterization hardware, ray tracing scales trivially, and can easily be parallelized. Several approaches have already been realized on MIMD and SIMD architectures [4, 5, 13]. Exploiting this scalability by massive parallelization has recently allowed interactive ray tracing to be achieved in software. It was first realized on supercomputers [16, 12, 19, 17, 18, 23] and more recently, interactive performance has been brought to commodity clusters of standard PCs [30, 31]. Our hardware implementation is based on an extended version of the software approach outlined in [30]. For a detailed overview of the state-of-the-art in interactive ray tracing see [27].

Beside using existing architectures, several special purpose hardware architectures for ray tracing have been developed. Initially hardware support provided was only for the intersection-operation. For a survey see [3]. Later DSPs were used to build PC-card based ray tracing accellerators [6]. Several volume ray casters on PC-cards have been developed [15, 22, 21] and there is a commercially available hardware architecture for high quality off-line ray tracing [26, 7].

While these projects achieved remarkable results and work well as accellerators for ray tracing, none of them are capable of delievering full-screen real-time frame rates comparable to those of current rasterization hardware.

Instead of designing special purpose hardware for ray tracing, another interesting approach is to map it to more general architectures available in the near future: Using a a multi-processor system on a single chip [14], Purcell has developed a highly optimized distributed ray tracer that would be capable of delivering interactive performance [24]. More recently, it has been shown that ray tracing can also be mapped to next generation's standard rasterization hardware using shader programs [25]. These two projects show that ray tracing can in principle be realized on such hardware architectures. In this paper we explore the efficiency and complexity required for designing special purpose hardware for ray tracing.

3. The Ray Tracing Algorithm

The ray tracing algorithm consists of three parts: generating rays from the viewer's eye through the pixels of the screen, tracing these rays through the scene delivering triangle hit points, and finally shading the ray given a hit point. For advanced lighting effects new rays may be spawned recursively from the hit point.

In this paper we concentrate on the second part: tracing rays through the scene. This has been the major bottleneck in previous attempts to design hardware for ray tracing. Issues are dealing with the recursion and reducing the bandwidth requirements to the cache, which for a naive implementation would reach $\approx$ 300 GB/s at 100 fps for our benchmark scenes.

[†] measured in floating point power and bandwidth to main memory

3.1. Tracing Packets of Rays

Wald et al.[30] have proposed a simple iterative traversal algorithm using axis-aligned BSP trees and a fast ray triangle intersection routine. In their system rays are traced in packets of four rays, which reduces bandwidth and allows the use of SIMD instructions available in modern processors.

We use the same approach but with larger packets of rays. All rays of a packet are traversed simultaneously through the BSP tree, entering a child node if any of the rays pierces it. In the case when both children need to be traversed, the iterative traversal algorithm stores one of the children onto a stack and enters the other one. Obviously, grouping rays in a packet only works out if the rays are coherent and visit roughly the same items in the BSP tree. Then traversed BSP nodes need to be fetched only once to be used for every ray in a packet. This dramatically reduces the bandwidth in proportion to the size of the packet.

However, large packets also increase the overhead as they cause more rays to traverse BSP nodes they would not traverse if traced individually or in smaller packets. To overcome this problem, we associate a bit vector with each packet. For each ray this bit vector indicates whether the ray is active in the current branch of the BSP tree. This allows us to efficiently operate on only a subset of rays in a packet and dramatically reduces the overhead. In hardware the overhead of updating and evaluating the bit vector is almost negligible.

Measurements show that groups of 64 rays are a good compromise between bandwidth requirements, additional overhead, and on-chip memory required for storing the rays. Figure 1 shows the influence of the number of rays per packet on the amount of memory transfered during rendering of one frame (i.e. the bandwidth to the cache). It also shows the average amount of memory touched by tracing one packet. This amount gives a rough estimate of the minimum size a cache should have.

4. The Hardware Architecture

Our hardware architecture (see Figure 2) consists of a custom ray tracing chip connected to several SDRAM chips, a framebuffer, and a PCI/AGP bridge all placed on a single PC board. The PCI/AGP bridge is used to upload scene data and camera settings from the host. The SDRAM chips store the entire scene including geometry, BSP tree, and materials/shaders. The image is rendered into the framebuffer or transfered back to the host.

The custom chip contains our ray tracing hardware architecture *SaarCOR* (Saarbrücken's Coherence Optimized Ray Tracer).

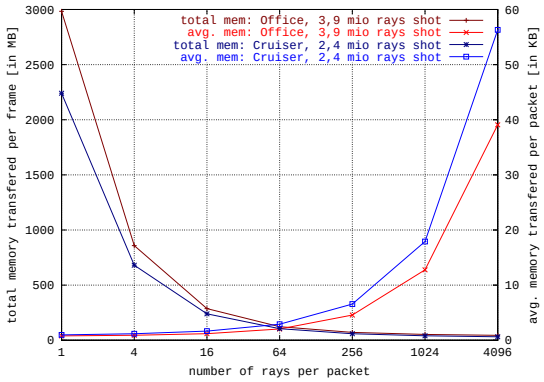

Figure 1: *Influence of the number of rays per packet on the amount of memory transfered during rendering of one frame and the average amount of memory requested by a packet. The latter figure gives a rough estimate of the minimum size a cache should have.*

4.1. The SaarCOR Chip

The SaarCOR chip is split into three main units: the ray generation and shading unit (*RGS*), the ray tracing core (*RTC*), and a unit to manage memory access (*RTC-MI*).

The architecture is designed to be modular and scalable by exploiting the inherent parallelism of ray tracing and shading. Adding more functional units to either the RGS or the RTC allows to independently scale the shading or ray tracing performance, respectively.

4.1.1. Ray Generation and Shading

The ray generation and shading unit is again split into three types of sub-units: a single *master* determining which eye ray will be rendered next, one *slave* for each connected RTC receiving the coordinates of a pixel from the master and managing this ray until it is fully rendered, and the unified memory interface (*MemInt*) handling all memory accesses. In order to scale with the shading complexity each slave can contain several shading units.

4.1.2. Ray Tracing Core

The ray tracing core (RTC) traces rays through the BSP acceleration structure and intersects rays with triangles found in the leaf nodes. According to the basic structure of ray tracing, the RTC is again split into three sub-units. The *traversal unit* receives rays from the RGS and traces them until it locates a BSP node containing a list of triangles. The list address is then forwarded to the rather simple *list unit* fetching the addresses of the triangles and sending their addresses to the *intersection unit*.

The intersection unit finally fetches the triangle data and performs the intersection computation. The results of these intersections are then sent back to the traversal unit. Depending on the intersection results it continues tracing the rays or

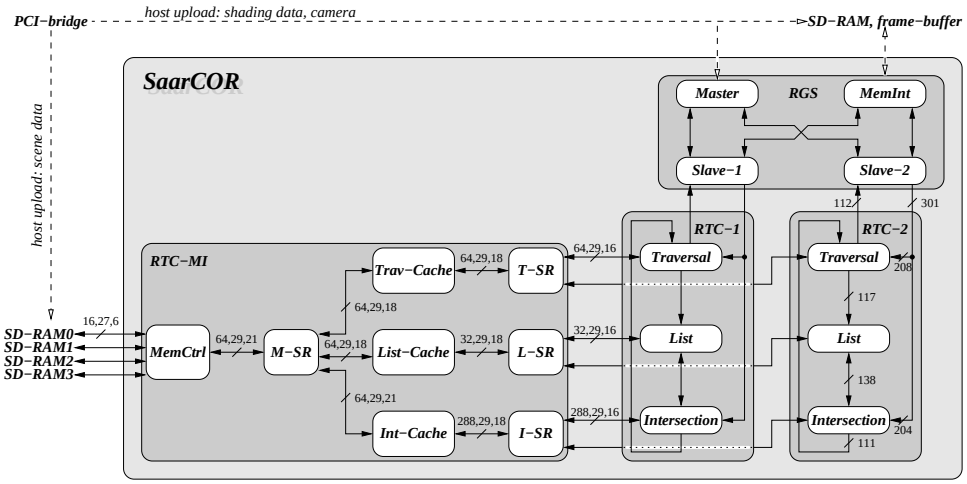

Figure 2: *The SaarCOR hardware model splits into three parts: The ray tracing core (RTC), the ray-generation and shading unit (RGS), and RTC's memory manager (RTC-MI). To allow for easy scaling, several RTCs are supported. Please note the simple routing scheme used: it contains only point-to-point connections and small busses, whose width is also shown separated into data-, address- and control-bits.*

sends the final results back to the slave and its shading unit for further processing. Each set of RTC and RGS-slave in such a configuration is completely independent except for the memory interface and the connection to the RGS-master.

4.1.3. RTC Memory Interface

The memory interface for the ray tracing core (*RTC-MI*) handles memory requests for all cores. It consists of several simple routing units (*SR*) implementing a simple but efficient routing scheme, three caches each holding a different type of data, and the external memory controller (*MemCtrl*) connecting to the SDRAM chips.

4.2. Implementation Issues

In the following we analyze the requirements of an efficient and high-performance ray tracing implementation and discuss possible configurations that fulfill these requirements. We also discuss potential problems and optimizations.

For each bus Figure 2 lists its width separated in the number of data, address, and control bits. It becomes obvious that the interconnection between the units and sub-units is very simple and narrow, and does not require complex routing. We exclusively use point-to-point links simplifying the overall design and its implementation in hardware.

The architecture is designed to be fully pipelined and careful attention has been paid to avoid stalls in any parts of the system. By avoiding stalls we can build deeply pipelined units without running into performance issues (see Section 5).

4.2.1. Ray Tracing Core

The ray tracing core consists of three sub-units: the traversal, the list, and the intersection unit. The implementation of the list unit is straight forward and not discussed any further.

We start with an estimation of the cost of traversal operations (*trav-op*) and intersection operations (*int-op*). For a detailed overview of the basic algorithms see [30]. Each trav-op requires 64 bits of data, three floating point (FP) additions, and one FP multiplication. In contrast an int-op accesses 288 bits of data, and executes 12 FP adds and 13 FP multiplications (we assume the same cost for addition and subtraction and count a division as three multiplications).

This cost difference suggests that we should be able to trade intersection operations for traversal operations. This can be realized by using a finer space partitioning through a deeper BSP-tree, generally resulting in less triangles per leaf cell. While this can reduce the number of intersections by building tighter enclosures, it increases the average number of traversal operations required to locate triangles. Figure 3 shows this relation for a simple and a complex scene (see also Section 5).

We can exploit this relation in two ways. While planning the chip we can optimize for the average traversal to intersection cost of our target scenes. Secondly, for a given hardware we can load balance between the traversal and intersection units by varying the depth of the BSP tree. For rasterization hardware, load balancing between vertex and fragment processing can only be achieved with LOD techniques. However, generating LODs is a non-trivial task which must be supported by the application. On the other hand the BSP gen-

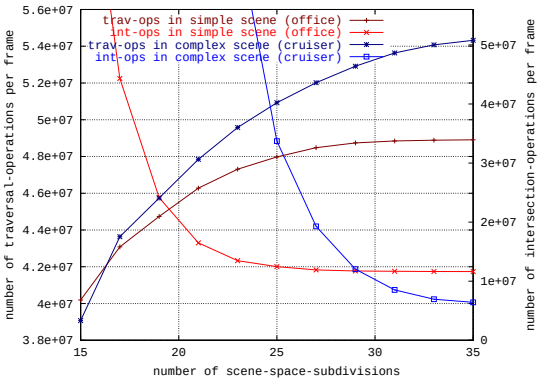

Figure 3: *Trading trav-ops vs. int-ops: given any cost-ratio between trav- and int-ops, there is an optimal number of scene-space-subdivisions. However, if this number is not optimally choosen, performance degrades only slowly.*

eration is fully automatic and can be handled in hardware for arbitrary scenes.

Our measurements show that a ratio of four traversal operations to one intersection operation is well suited for most scenes.

By supporting different numbers of traversal and intersection units we need to manage the communication between them. An important goal in our design was to keep this sort of communication as simple as possible. Having independent traversal units each traversing separate packets would require complex routing and scheduling schemes. Instead we statically split each packet of rays into as many groups as we have traversal sub-units.

Each traversal sub-unit only traverses its group of rays in a packet. The results of all sub-units are then combined to form the result that would have been computed by a single traversal unit. Thus they appear as a single traversal unit simplifying connections to the list and intersection units. This allows us to use direct point-to-point connections between units.

The overhead introduced by splitting the packet of rays statically grows with the number of groups or traversal sub-units. A group of rays scheduled to a traversal sub-unit may not contain any active rays making it run idle. However, in the case of four traversal sub-units running synchronously their average usage is still well above 90%.

Instead of having all traversal sub-units running synchronously, we can allow them to run asynchronously thus reducing the overhead introduced by a group of idle rays to a single cycle, independent of the number of traversal sub-units.

Another positive effect of using multiple traversal sub-units in parallel is the proportional reduction of latency caused by the traversal computation. The average latency of processing rays must be minimized in order to reduce on-chip storage requirements and to increase responsiveness.

Another source of latency is due to memory access. Any ray in a packet that requires access to slow external memory would stall the entire packet and reduce efficiency. To overcome this problem we operate on several packets of rays simultaneously. While one packet is being traversed, the data for the next one can be fetched, and another packet could be intersected. This concept is similar to *multi-* or *hyper-threading*, used on some of the latest processors from Intel and IBM and in parallel computers[20] to overcome memory latency and scheduling issues.

While a Pentium-4 is limited to two threads that increase the performance by up to 30%[9], we can do much better. Using 8 threads increases performance by 750% and with 16 threads still achieve 1220% for typical scenes. This indicates that we can hide almost all latency by using multi-threading allowing us to use deeply pipelined units and relativly slow memory.

Multi-threading with packets of rays may introduce bandwidth, latency, and bus width problems for transferring packets between units in a RTC. Instead of forwarding entire packets of rays during the computation, we initially broadcast the read-only part of rays to the traversal and intersection units, which keep local copies. During processing it is then sufficient to pass small ids for packets together with dynamic data, such as the current active bit-vector.

4.2.2. RTC Memory Interface

The RTC-MI connects all ray tracing cores with main memory. Since several units need to access a common bus, a routing scheme has to be used. However, complex schemes like butterfly networks are very costly to implement for large busses. Our goal was to simplify the architecture and avoid complex routing as far as possible. This leads to the most simple scheme using a round-robin multiplexer for submitting memory requests and a labeled broadcast to return data. Each unit allows for several outstanding memory requests.

Even though this scheme is extremely simple it fully satisfies the requirements of our hardware architecture, as the use of packets of rays has already reduced the bandwidth to the cache drastically. Obviously this scheme will break if too many RTCs increase the bandwidth requirements to the caches beyond the bus capacity. Our simulations show that this would happen with 8 to 16 RTCs, based on packets with 64 rays and depending on the scene. The scalability could be increased further by using larger packets of rays or by adding caches. Since all memory accesses are read-only, no cache consistency problems arise.

The bandwidth between the cache and main memory is rather low since even for small caches of 64 KB to 144 KB, depending on its data type, we achieve a cache hit rate of about 95%. For the caches we simulated direct-mapped, two,

and four way set-associative cache designs. However, performance was hardly effected by the cache design ($< 3\%$). These low bandwidth requirements allow us to use standard, low-cost, and simple PC133 SDRAM technology.

In order to provide scalability in the memory interface when multiple RTCs are used, we support several SDRAM chips connected to the memory interface and use simple address hashing to avoid hot spots. For our simulations (see Section 5) we assume that our SaarCOR chip runs at 533 MHz, giving a 4-to-1 clock ratio between the chip and memory. Depending on the benchmark scene, between one and four 16 bit wide SDRAM chips are required, offering a total memory bandwidth of 250 MB/s to 1 GB/s.

Currently our design requires all scene data to be stored in local RAM on the graphics card. In a related publication [31] we show how the memory accessible by the ray tracer could be used as a cache by dynamically loading missing scene data. We believe that a similar system would also work in hardware.

4.2.3. Ray Generation and Shading

In this paper we can only cover the ray traversal and intersection part of the rendering architecture. For completeness we provide a rough overview of shading. To this end we limit the shader to simple Phong-like shading with bilinear texture filtering. Its cost per ray is conservatively approximated with 50 FP additions and 70 FP multiplications including address calculation for texture reads.

Our simulations show that depending on the scene, a RTC finishes a ray every 20 to 80 cycles on a standard SaarCOR chip. By using pipelining or parallelization this requires 3 FP adders and 4 FP multipliers for each slave performing the shading operation. This makes it 4 to 8 times less expensive than the RTC unit in terms of hardware ressources.

Since shading is also performed in packets and no overdraw due to the Z buffer must be accommodated, memory bandwidth requirements for shading are comparable to that of the RTC even with bilinear texture access. Please note that for shading computations ray tracing will generally need significantly lower texture bandwidth than rasterization, where complex shading must often be precomputed and stored in textures. With asynchronous shading some of these computations can easily be performed on the fly saving texture memory and bandwidth.

A similar argument holds for coherence in texture access. As triangles get smaller objects space coherence exploited in rasterization is lost between many separately rendered triangles. Because we shade packets of rays we are largely independent of individual triangles and exploit coherence of textures as they are projected into image space.

With ray tracing shading parameters must not be carried along through the pipeline for every triangle but can be fetched once we know that we need to shade. Some of the shading parameters might even be shared between many triangles and must only be loaded once. This becomes increasingly important as more complex shaders are being used. Finally, shading is decoupled from visibility computations and allows the architecture to be tuned for specific target markets.

During rendering, several secondary rays (e.g. for shadows and reflections) may be generated. If more rays are generated than can be handled or stored by the hardware, we simply throw away partially completed rays to avoid deadlocking. Even this rather simple and naive solution reduces the performance by only 5% to 10% and leaves much room for further improvements.

5. Results and Discussion

The modular design of our architecture offers many possible configurations for a hardware ray tracer, which makes it difficult to present an exhaustive analysis. We solve this problem by presenting a *standard* configuration, which was derived from our set of benchmark scenes. For these scenes it is a good compromise between performance and hardware cost. We believe that this standard configuration is economically feasible even for a production-type chip with more complex shading, since this standard configuration requires less hardware ressources than current graphics chips based on rasterization (see below). We also present performance figures achievable for much larger hardware configurations in order to show the potential of our architecture.

Taking this standard configuration as the base line we then evaluate the impact of selectively changing one or a few parameters. The resulting figures should provide sufficient insight into the properties of the overall architecture.

The standard SaarCOR system consists of four RTCs each using 16 threads and four traversal sub-units. We assume that the chip runs at a clock rate of 533 MHz and is connected to four SDRAM chips running at 133 MHz via three caches of 272 KB total. This cache is split into 64 KB for the traversal cache, 64 KB for the list cache, and 144 KB for the intersection cache. Each cache stores items of different data types and sizes. For example, the intersection cache stores 4096 triangles of 288 bits each (see Figure 2).

With the standard configuration, a SaarCOR chip requires a total of 192 floating-point units, 822 KB for registers-files, and 272 KB for cache, adding up to 1094 KB total on-chip memory. All on-chip memory is split into small local pieces of memory, allowing for simple connections, and a feasible chip design. The RTCs are limited to a bandwidth of 1 GB/s to main memory. 192 FP-units is roughly half the hardware floating-point budget of state-of-the-art rasterization hardware. We assume that for a production-type chip, the other half would be spent on more complex shading. The floating-point units are streamlined versions not supporting

the full IEEE standard. For comparison: the GeForce3 has 380 floating-point-units and 7.2 GB/s of memory bandwidth.

5.1. A Simulator for SaarCOR

In order to analyze and evaluate our architecture, we performed cycle-accurate simulations of our design. The simulator can be adapted to a chip technology by specifying the number of gates that can be executed within one clock cycle. Thus the latency of a fully pipelined functional unit measured in cycles is its delay measured in gates divided by this technology constant. In this paper, we assume this constant to be four for all functional units.

In order to verify the correctness of the hardware simulation, we compared the traces with a separate instrumented software ray tracer.

With our simulator we achieve a run-time of roughly two hours per frame for a typical scene on a standard SaarCOR chip. This allows us to examine a wide range of system-parameters without being limited by simulator run times.

5.2. Test Environment

To cover a wide range of applications Table 1 provides examples with corresponding images shown in Figure 4. All scenes were tested at full-screen resolution of 1024x768 without oversampling. Please note that all lights in the scenes cast shadows.

Figure 4: *Some of the scenes used for benchmarking*

We group the scenes into three sections: the OpenGL-like, where only eye rays are shot, scenes with light sources and shadows, and scenes with light sources, shadows and multiple reflections.

The Quake3 scene consists of the level q3dm7 of the game Quake3-Arena[11]. The Sodahall scene gives a perfect example of a large seven-stories building completly modeled in high detail — chairs, books, plants, and even pencils on the desks are modeled. This model is highly occluded, where at each location only a small part of the scene is actually visible. This is where the built-in occlusion-culling of ray tracing performs optimally. The CruiserGL scene models a large part of a navy battle cruiser in very fine detail.

scene	#triangles	#lights	reflection-depth	rays shot
Quake3	34 772	0	0	786 432
Sodahall	1 510 322	0	0	786 432
CruiserGL	3 637 101	0	0	786 432
Conf	282 000	2	0	2 359 296
Cruiser	3 637 101	2	0	2 359 296
Office	33 952	3	3	3 863 846
BQD-1	2 133 537	1	3	1 583 402
BQD-2	2 133 537	1	3	1 548 632

Table 1: *The scenes used for benchmarking*

The Conf scene is a model of a conference room with many chairs and two light sources. The Cruiser model is similar to CruiserGL, but enhanced with two light-sources.

The Office scene contains three light-sources and several reflective objects, like the window and the reflective ball. The BQD scene is another game-like model. It consist of a large terrain with the Quake3 scene placed in a valley. Furthermore it was enhanced by a parallel light source modeling the sun. Inside the Quake3 scene is a reflective teapot and a column giving an idea of effects possible with ray tracing. While in the first view we stand inside the building, BQD-2 is a view from above. Please note that no level-of-detail mechanism has been used to approximate far-away geometry.

5.3. Performance Measurements

Table 2 presents the performance achievable with the standard SaarCOR chip.

scene	1 RTC	2 RTCs	4 RTCs	
Quake3	27.20	54.45	111.12	fps
Sodahall	28.88	56.71	113.22	fps
CruiserGL	28.58	52.04	65.86	fps
Conf	8.91	16.77	31.56	fps
Cruiser	9.82	17.38	20.05	fps
Office	7.52	14.34	28.56	fps
BQD-1	11.74	23.12	45.90	fps
BQD-2	7.55	12.98	17.43	fps

Table 2: *Absolute performance measurements for the SaarCOR chip with 1, 2 and 4 RTCs, 272 KB cache, and 1 GB/s memory bandwidth. 4 RTCs have only half the floating-point performance of a GeForce3 and there is an almost linear relation between performance and the number of RTCs.*

The performance measurements of Table 2 show several

interesting points: The performance scales almost linearly with the number of RTCs used and with the number of rays used to calculate the image (see Table 1). In comparison to rasterization, where performance degrades linearly with the number of triangles in the scene [30], this number has only a small impact on the performance for our architecture. However, some figures are not as expected. In particular both Cruiser-scenes and BQD-2 show that there must be a bottleneck limiting the performance of the system.

A closer analysis shows, that the Cruiser scene with 3.5 million triangles is limited by the memory bandwidth for triangle fetching. Table 3 gives performance measurements of the CruiserGL scene for different sized intersection-caches in combination with 1 and 2 GB/s bandwidth to main-memory. This shows that with a bandwidth of 2 GB/s and an int-cache of 288 KB the performance again scales linearly in the number of RTCs. Achieving linear speed-up with 4 RTCs in BQD-2-scene is harder: we need to enlarge all caches four times to roughly 1 MB together with a 2 GB/s bandwidth to main-memory.

size of int-cache	144 KB	288 KB	576 KB
1 GB/s (4 SDRAMs)	65.86 fps	77.54 fps	86.36 fps
2 GB/s (8 SDRAMs)	87.24 fps	103.62 fps	113.89 fps

Table 3: *Influence of memory bandwidth and size of the int-cache (which caches the triangles) on the scene CruiserGL with 4 RTCs. This shows again a linear speed-up with the number of RTCs.*

In contrast to these complex models, the Quake3 scene shows perfect linear scaling. Using the standard cache and a bandwidth of only 250 MB/s linear scaling is achieved even up to 16 RTCs. The floating-point performance of the GeForce3 equals the floating-point performance of a full SaarCOR chip with 8 RTCs and full shading. Rendering the Quake3 scene with 8 RTCs achieves 235 fps.

Further analysis leads to the observation that for most scenes linear speed-up with n RTCs can be achieved by using n SDRAMs, resulting in a bandwidth of $n \times 250$ MB/s.

The performance of a chip can be measured in two ways: the absolute and the relative performance. Table 2 lists the absolute performance, while Table 4 shows the relative performance. The relative performance is defined as the percentage of absolute performance versus ideally achievable performance. The ideally achievable performance is defined as

$$\text{fps}_{ideal} = \frac{\text{chip-speed in cycles per second}}{max\{\frac{\#\text{trav-ops}}{\#\text{RTCs}\times\#\text{trav-sub-units}}, \frac{\#\text{int-ops}}{\#\text{RTCs}}\}}$$

Simply speaking: if there is no overhead at all, we need at least one cycle for every operation we have to perform. If we divide the number of operations by the number of units we obtain the theoretical achievable minimal number of cycles needed.

Table 4 shows that even the simple architecture of the standard SaarCOR chip already achieves 70%–80% of the ideal performance. Using 32 threads instead of 16 threads per RTC, we increase these results by 10% achieving 80%–90% of the ideal performance. On the other hand, using 32 threads instead of 16 increases the on-chip memory from 822 KB to 1050 KB (not counting the caches). If we increase the size of the cache, the memory-bandwidth or the number of threads per RTC, these figures can be improved even further. So depending on the price one is willing to pay, nearly arbitrary figures can be achieved. This shows the flexibility of ray tracing and our hardware architecture, which can be scaled over a wide performance range.

scene	1 RTC	2 RTCs	4 RTCs
Quake3	76%	76%	78%
Sodahall	80%	79%	79%
CruiserGL	71%	65%	41%
Conf	67%	63%	59%
Cruiser	72%	63%	37%
Office	71%	68%	68%
BQD-1	73%	72%	71%
BQD-2	54%	46%	31%

Table 4: *Relative performance: percentage of the theoretically ideal performance achieved with a standard SaarCOR chip*

The relative performance, as listed in Table 4, equals roughly the usage of the traversal and intersection units. Let c be the number of clock-cycles for rendering an image and w the number of cycles a unit or a bus was busy. We then define the usage as w/c. Figure 5 shows several characteristic measurements for a standard SaarCOR chip running the BQD-1 scene: a) The usage of each of the traversal sub units, b) the usage of each of the traversal and intersection units, c) the percentage a unit contributed to all accesses to the common bus, d) the usage of the bus to the caches, e) the hit rate of the caches, f) the percentage a cache contributed to all accesses to memory, g) the usage of the bus to the memory controller, h) the percentage of all accesses to the memory each SDRAM handles.

The high amount of traffic to main-memory contributed by the int-cache is due to the fact that all accesses to main-memory are only 64 bits wide and therefore each triangle requires 5 consecutive accesses.

5.4. Influence of the Accelleration Structure

As mentioned in Section 4.2.1, the number of scene subdivisions of our accelleration structure can be used to adjust the hardware architecture to any scene and vice versa. Changing the number of subdivisions influences the architecture in three ways: As shown in Figure 3, the number of traversal and intersection operations required to calculate a frame

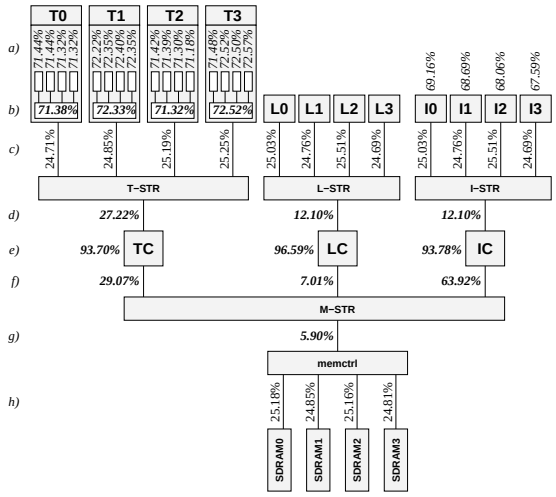

Figure 5: *Usage and hit-rates of a standard SaarCOR chip running the BQD-1 benchmark: this shows in detail, that trivial static load-balancing works out perfectly well*

changes, resulting in different frame rates. As the number of scene-space-subdivisions increases, the memory needed to store all items of the BSP grows exponentially. Since the iterative traversal of a BSP-tree requires a stack of the size of the maximum depth of the scene subdivisions, the required on-chip memory increases linearly with the number of subdivisions.

The following formula calculates the on-chip memory of a standard SaarCOR chip depending on the number d of scene subdivisions:

$$\textit{on-chip-memory} = \textit{cache} + 287.6\ \text{KB} + d \times 17.25\ \text{KB}$$

The algorithm we used to build the BSP is a very simple one. First results on several more advanced algorithms show that there is much room for improvements with regards to rendering performance and BSP memory.

5.5. Lights, Reflections, and Anti-Aliasing

One of the main advantages of ray tracing is its ability to render physically correct shadows, reflections, and refractions. In this section we analyse the impact of these different types of rays on the overall performance by rendering the Office scene in different conditions, as listed in Table 5 and shown in Figure 7 (in the color section): (a) eye rays (er) only, (b) er and reflections up to 3 levels (r3), (c) er and 3 lights (3l), (d) er, reflections and 3 lights, (e) er with a simple four times oversampling (4×os), i.e. for each pixel, 4 rays are shot and their contribution is averaged to calculate the color of the pixel. Please note that in (b) 20% of all rays are reflected.

Table 5 shows that the performance degrades linearly with the number of rays shot, independently of the type of rays. This is also true for refracted rays used to simulate glass-effects (not shown here). Case (e) shows that oversampling

	#rays	$\frac{\#rays(er)}{\#rays}$	FPS	$\frac{fps}{fps(er)}$
(a) er	786 432	100%	127.75	100%
(b) er,r3	966 275	81%	99.23	78%
(c) er,3l	3 145 728	25%	36.67	29%
(d) er,r3,3l	3 863 846	20%	28.56	22%
(e) er,4×os.	3 145 728	25%	35.06	27%

Table 5: *Office with different types of rays. This shows that the performance is very close to linear in the number of rays shot and almost independent of the type of the ray.*

is slightly cheaper than linear: 4 times more rays cost only 3.6 times more, due to a better cache hit-rate. See Table 6 for a detailed look on the cache.

oversampling	none	4-times
hit-rate trav-cache	89.9%	96.8%
hit-rate list-cache	89.7%	95.7%
hit-rate int-cache	97.1%	98.8%

Table 6: *By using simple four-times oversampling the cache-hit-rate increases, giving a 10% performance improvement over the expected cost of anti-aliasing.*

6. Conclusions and Future Work

In this paper we have presented a flexible, modular, and scalable hardware architecture for real-time ray tracing. At costs comparable to current rasterization chips it offers similar performance with the same type of OpenGL-like scenes. In addition it offers all the benefits of ray tracing including accurate shadows, correct reflection, refraction, and built-in occlusion culling.

Using extensive cycle-accurate simulations we evaluated the properties and performance of the architecture for a wide set of test scenes. The results show that the architecture scales well in the number of functional units used.

In this paper we concentrated on the visibility computations and simple shading, but the simulations indicate that support for advanced shading would change little in the basic architecture and performance results.

With the flexible and modular design, our architecture can be configured to support a wide range of applications and cost-performance ratios. Even for a fixed architecture load balancing can be improved through properly built BSP trees.

We have shown that a simple approach using only static load balancing, trivial routing, low memory bandwidth, simple memory technology, and small caches is sufficient for achieving these results. This is promising as it leaves many opportunities for later optimizations and extensions.

The architecture as presented here is able to support a visual quality at the level of standard OpenGL plus standard ray tracing features. Further work is required to include advanced and programmable shading, as well as dynamically changing scenes. More work is also required for evaluating

the architecture with respect to specific VLSI technolgies such as FPGAs or ASICs. A key factor for the success of ray tracing will be the API issue, where initial results are available [29].

In summary, it seems that the old dream of real-time ray tracing is finally realizable at hardware costs similar to existing graphics systems. This would enable the display of highly realistic, physically correct, and accurately lit interactive 3D environments. Because ray tracing is at the core of any algorithm computing light transport, fast ray tracing is likely to also enable real-time global illumination computations and other advanced optical effects.

References

1. Kurt Akeley. RealityEngine graphics. In *Computer Graphics (ACM Siggraph Proceedings)*, 1993. 2
2. Matthew Eldridge, Homan Igehy, and Pat Hanrahan. Pomegranate: A fully scalable graphics architecture. *Computer Graphics*, pages 443–454, July 2000. 2
3. Stuart A. Green. Parallel processing for computer graphics. *MIT Press*, pages 62–73, 1991. 2
4. Stuart A. Green and Derek J. Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications*, 9(6):12–26, 1989. 2
5. Stuart A. Green and Derek J. Paddon. A highly flexible multiprocessor solution for ray tracing. *The Visual Computer*, 6(2):62–73, 1990. 2
6. C. Scott Ananian Greg Humphreys. Tigershark: A hardware accelerated ray-tracing engine. Technical report, Princeton University, 1996. 2
7. D. Hall. The AR350: Today's ray trace rendering processor. In *Proceedings of the Eurographics/SIGGRAPH workshop on Graphics hardware - Hot 3D Session 1*, 2001. 2
8. Pat Hanrahan and Jim Lawson. A language for shading and lighting calculation. In *Proceedings of SIGGRAPH*, 1990. 1
9. http://developer.intel.com/technology/hyperthread. Introduction to hyper-threading technology, 2002. 5
10. http://www.nvidia.com. Geforce3 - the world's most advanced processor, 2001. 2
11. http://www.quake3arena.com/. Id-software: Quake3-arena, 2001. 7
12. M. J. Keates and Roger J. Hubbold. Interactive ray tracing on a virtual shared-memory parallel computer. *Computer Graphics Forum*, 14(4):189–202, 1995. 2
13. Tony T.Y. Lin and Mel Slater. Stochastic Ray Tracing Using SIMD Processor Arrays. *The Visual Computer*, pages 187–199, 1991. 2
14. K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart Memories: A Modular Recongurable Architecture. *IEEE International Symposium on Computer Architecture*, 2000. 2
15. M. Meissner, U. Kanus, and W. Strasser. VIZARD II, A PCI-Card for Real-Time Volume Rendering. In *Eurographics/Siggraph Workshop on Graphics Hardware*, 1998. 2
16. Michael J. Muuss. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium '95*, June 1995. 2
17. 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), 1999. 2
18. Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive ray tracing for isosurface rendering. In *IEEE Visualization '98*, 1998. 2
19. Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive ray tracing. In *Interactive 3D Graphics (I3D)*, pages 119–126, April 1999. 2
20. Wolfgang J. Paul, Peter Bach, Michael Bosch, Jörg Fischer, Cédric Lichtenau, and Jochen Röhrig. Real PRAM-Programming. In *Proceedings of EuroPar 2002*, 2002. 5
21. Hans-Peter Pfister. SIGGRAPH course on Interactive Ray Tracing, 2001. 2
22. Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The VolumePro real-time ray-casting system. *Computer Graphics*, 33, 1999. 2
23. Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. *Computer Graphics*, 31(Annual Conference Series):101–108, August 1997. 2
24. Timothy Purcell. The SHARP Ray Tracing Architecture. SIGGRAPH course on Interactive Ray Tracing, 2001. 2
25. Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray Tracing on Programmable Graphics Hardware. In *Proceedings of SIGGRAPH 2002*, 2002. 2
26. Advanced Rendering Technologies. http://www.art.co.uk/, 2002. 2
27. I. Wald and P. Slusallek. State-of-the-Art in Interactive Ray-Tracing. In *State of the Art Reports, EUROGRAPHICS 2001*, pages 21–42, 2001. 2
28. Ingo Wald, Carsten Benthin, and Philipp Slusallek. A Simple and Practical Method for Interactive Ray Tracing of Dynamic Scenes. Technical report, Computer Graphics Group, Saarland University, http://www.openrt.de, 2002. 2
29. Ingo Wald, Carsten Benthin, and Philipp Slusallek. OpenRT – A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. Technical report, Computer Graphics Group, Saarland University, http://www.openrt.de, 2002. 2, 10
30. Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, 20(3), 2001. 1, 2, 3, 4, 8
31. Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. In *Proceedings of the 12th EUROGRPAHICS Workshop on Rendering*, June 2001. London. 2, 6

Figure 6: *Some of the scenes used for benchmarking the SaarCOR architecture. It shows that for an appropriatly choosen cache size and bandwidth to memory the performance of the architecture scales very well in the number of processing units used and nearly independend of the number of primitives in a scene. See Tables 2,3 and 4 for details.*

Figure 7: *The Office scene with (from left to right): eye rays only (er); er and reflections; er and three point lights; er, reflections, and three point lights. It shows that the performance scales linear in the number of rays shot and independent of the type of the ray, i.e. eye, reflection and shadow rays have roughly the same cost. See Tables 5 and 6 for details.*