Estimating Performance of a Ray-Tracing ASIC Design

Sven Woop[*] Saarland University
Erik Brunvand[†] University of Utah
Philipp Slusallek[‡] Saarland University

Figure 1: Test scenes used to evaluate the DRPU ASIC: *Conference* (282k triangles) , *Mafia* (15k triangles), *Skeleton* (16k triangles), *Helix* (78k triangles), and *DynGael* (85k triangles). For more test scenes see Figure 6.

Abstract

Recursive ray tracing is a powerful rendering technique used to compute realistic images by simulating the global light transport in a scene. Algorithmic improvements and FPGA-based hardware implementations of ray tracing have demonstrated realtime performance but hardware that achieves performance levels comparable to commodity rasterization graphics chips is still not available.

This paper describes the architecture and ASIC implementations of the DRPU design (Dynamic Ray Processing Unit) that closes this performance gap. The DRPU supports fully programmable shading and most kinds of dynamic scenes and thus provides similar capabilities as current GPUs. It achieves high efficiency due to SIMD processing of floating point vectors, massive multithreading, synchronous execution of packets of threads, and careful management of caches for scene data. To support dynamic scenes B-KD trees are used as spatial index structures that are processed by a custom traversal and intersection unit and modified by an Update Processor on scene changes.

The DRPU architecture is specified as a high-level structural description in a functional language and mapped to both FPGA and ASIC implementations. Our FPGA prototype clocked at 66 MHz achieves higher ray tracing performance than CPU-based ray tracers even on a modern multi-GHz CPU. We provide performance results for two 130nm ASIC versions and estimate what performance would be using a 90nm CMOS process. For a 90nm version with a 196mm^2 die we conservatively estimate clock rates of 400 MHz and ray tracing performance of 80 to 290 fps at 1024x768 resolution in our test scenes. This estimated performance is 70 times faster than what is achievable with standard multi-GHz desktop CPUs.

CR Categories: I.3.1 [Hardware Architecture]: Graphics processors; I.3.7 [3D Graphics and Realism]: Ray-Tracing

Keywords: Ray-Tracing, Hardware Architecture, ASIC Implementation, Performance Estimation

[*]woop@cs.uni-sb.de
[†]elb@cs.utah.edu
[‡]slusallek@cs.uni-sb.de

1 Introduction

The current state-of-the-art in realtime computer graphics is the rasterization algorithm, mainly because low cost and highly efficient hardware implementations are available that achieve remarkable levels of performance. The basic principle of this algorithm, used in all current commodity graphics chips, is to *independently* rasterize one triangle at a time onto the screen. This local triangle operation can be computed quickly using deep pipelines of custom floating point hardware. However, the incorrect assumption that triangles are independent is the great weakness of rasterization as it limits the possible shading operations to local per-triangle computations. This does not allow for directly computing any global light effects such as shadows, reflections, transparency, or indirect illumination as this would require direct access to potentially the entire scene database during rendering. Multi-pass techniques that are often used to approximate these effects are inaccurate and inefficient, especially with respect to the required external memory bandwidth.

The trend in realtime computer graphics is towards high realism, which becomes more and more difficult to achieve with rasterization. Conceptually simple simulation-based rendering techniques like the ray tracing algorithm [2] can compute highly realistic images by *simulating* the physics of light based on the rendering equation. Recursive ray tracing [32] can easily compute shadows, reflections, refractions, and even combinations of them by recursively spawning secondary rays at the object intersection point. Ray tracing even allows global illumination to be computed by stochastically gathering incoming light at a point of interest [27]. The results of this computation are high quality, photo-realistic images that are often hard to distinguish from photographs.

Despite all these algorithmic advantages, ray tracing suffers from its high computational cost, causing renderings to take many seconds to hours to finish. Much research has been performed over the last two decades to speed up this computation, using different platforms and algorithms.

1.1 Previous Work

On the software side significant research has been performed on mapping ray tracing efficiently to parallel machines, including MIMD and SIMD architectures [7, 12]. The key goal has been to exploit the parallelism of the hardware architecture in order to achieve high floating point and thus high ray tracing performance [16, 15]. The OpenRT project implemented a high performance ray tracer for commodity PCs that are connected via a stan-

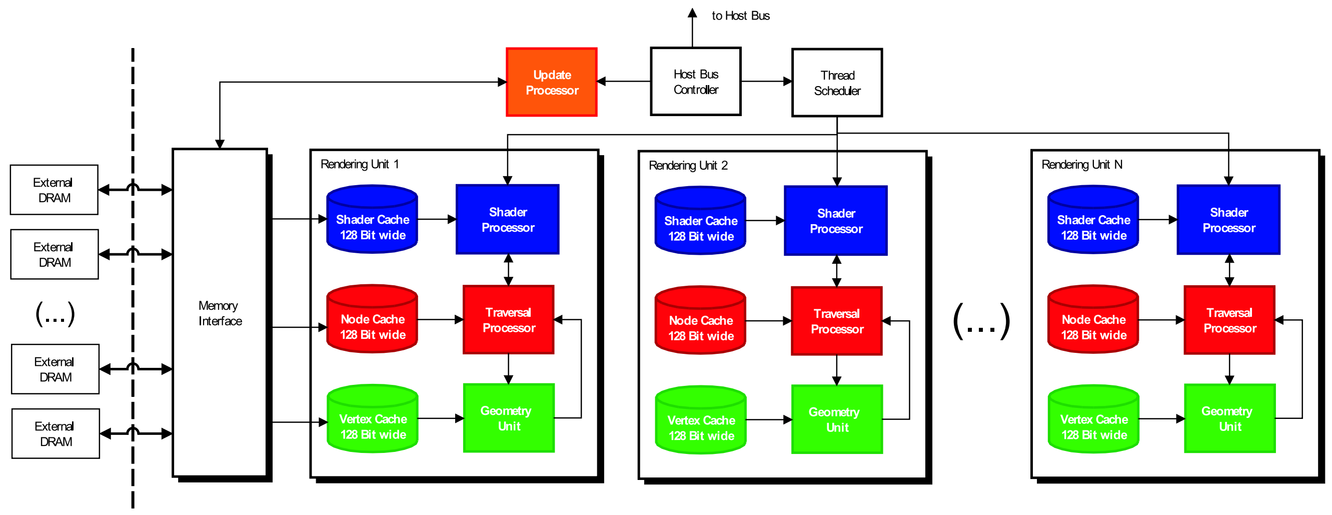

Figure 2: DRPU Architecture: Several Rendering Units per chip are supported by the DRPU architecture. These units consist of an application programmable *Shader Processor (SP)* to generate and shade rays and a fixed-function part that contains a high performance *Traversal Processor (TP)* to traverse the B-KD tree (when requested to by the SP) and a *Geometry Unit (GU)* to intersect rays with triangles or to transform them to the local coordinate space defined by a B-KD transformation node. These units are all connected to the Memory Interface via small first level caches and a Thread Generator schedules new pixels for computation. On dynamic scene changes the B-KD trees are efficiently updated by the *Update Processor*.

dard Ethernet network [30, 25]. We use this system for performance comparison.

Existing *programmable GPUs* available in the graphics cards of today's PCs can be used for many computationally intensive algorithms as they offer excellent raw floating point performance by implementing up to 48 SIMD processors. However, the programming model of these GPUs is very limited and does not efficiently support ray tracing [19, 5]. In particular, it does not provide flexible control flow and supports only very restricted memory access.

With realtime ray tracing it becomes also necessary to handle interactive changes in dynamic scenes. This is possible by using grids as spatial index structure as they allow for fast insertion of objects and even fast coherent rendering [29]. Separation of the scene into objects with piece-wise rigid motion and separate static spatial indices has been suggested [11] and has been implemented for realtime use on a cluster of PCs [26]. Bounding Volume Hierarchies [23, 28] have also successfully been used for rendering of dynamic scenes.

In recent years multiple *custom hardware* architectures for ray tracing have been proposed, both for volume [17, 8] and surface models. Partial hardware acceleration has been proposed [6] and a different implementation is commercially available [9]. In addition a complete ray tracing hardware architecture has been simulated [10]. The first complete, fully functional *realtime ray tracing chip* was presented in [20, 21]. With the RPU hardware architecture [35] a fully programmable design was implemented with limited support for dynamic scenes. A fixed function architecture using B-KD trees to render highly dynamic scenes was published in [34]. Although the results of these hardware implementations are promising, none of them achieves performance levels and functionality comparable with current rasterization hardware.

2 DRPU Hardware Architecture

The DRPU approach of this paper is the first one that supports programmable material and lighting shaders on the one hand and highly dynamic scenes on the other hand. The architecture mainly consists of two parts:

1. Ray Casting Units for managing spatial index structures during rendering and for manipulating them on scene changes

2. a Shader Processor which consists of four highly multi-threaded 4-way vector units (SPUs) for SIMD synchronous execution of bundles of threads to perform shading and ray generation tasks.

The Ray Casting Units are identical to the architecture as described in [34] while the Shader Processor (SP) is very similar to the SPUs described in [35]. A contribution of this paper is the combination of both designs, which makes it comparable to rasterization hardware in terms of support for programmable shading and handling of many kinds of dynamic scenes. This paper describes the basic details of B-KD trees and the SPU, while more details can be found in the related papers. The DRPU architecture also contains the Skinning Processor as described in [34] which is not explained further here.

The main contribution of this paper is that we implemented and tested the DRPU on an FPGA and especially recast that same architecture to an ASIC using a 130nm CMOS standard cell library from UMC [24]. We use the FPGA version to help calibrate the performance estimations of our ASIC implementation. We then extrapolate performance to a 90nm version, which shows that with a comparable amount of hardware resources as current GPUs one can achieve a comparable level of rendering performance, while gaining all the advantages of the ray tracing algorithm.

The DRPU hardware architecture is designed for ray tracing of dynamic scenes with programmable material and lighting shaders. It is highly scalable by supporting several Rendering Units on a single chip (see Figure 2). Each such Rendering Unit consists of a Shader Processor (SP) to shade packets of four rays, a Traversal Processor (TP) to traverse packets of four rays through B-KD trees, and a Geometry Unit (GU) to intersect packets of rays with triangles. To hide computation and memory latencies several of these packets of rays are processed in parallel in the highly multithreaded hardware. The threads are scheduled by a Thread Scheduler that performs load balancing and thread generation. Each time a packet of threads has finished its execution in one of the Rendering Units, the Thread Scheduler sends four new adjacent pixels to the Rendering Unit for processing. There, the packet of four threads is initialized and executed synchronously. The threads stay together in the packet, but may be masked out on diverging control flow. The SP, TP, and GU, each support the same number of thread-packets such that a packet can always continue its compuation in a different unit.

On dynamic scene changes the *Update Processor* on the chip is used prior to rendering to update bounds of the B-KD trees to adapt to the scene changes. The spatial index structure and the hardware units that handle it are explained in detail in the following section.

3 Ray Casting Units for Dynamic Scenes

For efficiently tracing rays through a scene, ray tracing requires spatial index structures that subdivide space into cells that can efficiently be enumerated along a ray. However, recomputing these spatial index structures is very expensive, which can limit ray tracing to static scenes. To cope with this problem we chose B-KD trees as index structure, which is a kind of Bounding Volume Hierarchy with one dimensional bounds. The structure of the B-KD tree is computed initially and maintained during rendering where only some node bounds need to be recomputed.

Such a B-KD tree is a binary tree, where each node recursively subdivides the geometry of the scene into two disjoint subsets represented by its two children. Each node stores the index of a coordinate axis and bounds on the geometric extent of its two children along this axis in the form of two bounding intervals often also referred to as slabs (see Figure 3). Each leaf node stores a reference to a single primitive of the scene. For instantiations of objects we support *transformation nodes*, that store a pointer to the objects root node, and a transformation matrix to specify its position.

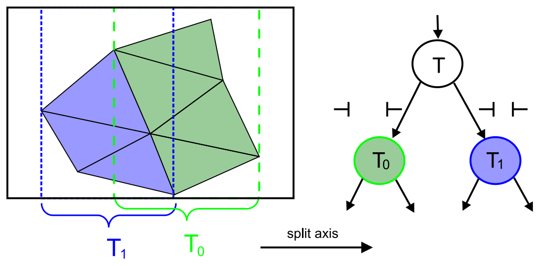

Figure 3: B-KD tree: A B-KD tree node divides a set of primitives into two disjoint subsets, represented by the two children. The node stores the extent of the geometry for each child as two bounding intervals along one splitting axis. The geometry is recursively subdivided until there is only a single primitive per node.

A main advantage of the B-KD trees is low memory consumption as they store the bounds of the children in only a single dimension. The implicit full bound per node can be obtained, similar to KD trees, by clipping against the bounding intervals from the top down. We build our B-KD trees using a Surface Area Heuristic (SAH) similar to the approach in [28]. The concept of B-KD trees is described in more detail in [34].

3.1 Update Processor for B-KD Trees

For changed geometry the B-KD tree bounds can be updated by a simple bottom-up algorithm that merges the full axis aligned bounds of the nodes from bottom-up through the tree and updates for each B-KD tree node the extent of the two children along the node axis. This algorithm can be implemented by only performing trivial min/max operations that do not touch the structure of the tree.

This update procedure is performed by a dedicated *Update Processor* that is fed by an instruction stream which is precomputed by the driver application for each dynamic object. This instruction stream includes instructions for loading vertices into one of the 64 *vertex registers*, computing a triangle bound from 3 vertices, and merging two bounds together. By operating on vertices the processor is optimized for triangle meshes with shared vertices. By keeping these shared vertices in the vertex registers they can optimally be reused for computing the bound of several triangles, thus no caches are required. All partial results, such as computed node bounds are stored to one of 64 special *bound registers* to minimize the external memory traffic to only the required updates of the nodes, vertex fetches, and additional instruction fetches.

For best results the structure of the B-KD tree should "match" the geometry and its dynamics. This means that geometry in a subtree should stay as close together as possible during the course of changes. A mismatch can result in significant overlap of the bounds of child nodes. This leads to redundant traversal and missed opportunities for early ray termination, as both child nodes *must* be traversed if a ray enters an overlap region. As a consequence only dynamic scenes that show some coherent motion can be handled efficiently with B-KD trees. Many typical motions, like skinned meshes, obey this restriction as will be shown in the result section by some animated characters. Random movement of triangles, however, is handled less efficiently because the significant overlaps would require the traversal of many B-KD tree nodes.

3.2 Traversal Processor (TP)

Traversing B-KD trees typically requires between 50 to 100 traversal steps. Using a fully programmable unit for these operations wastes precious cycles, since every step would correspond to several instructions. Instead a Traversal Processor (TP) is used that consists of four custom fixed function Traversal Processing Units (TPU) that are used in SIMD mode which greatly improves traversal performance as it can perform one packet traversal operation each clock cycle.

The Traversal Processor traverses several packets of four rays in parallel through the B-KD tree. In order to hide memory and computation latencies multiple packets of rays are processed simultaneously using a wide multi-threading approach [20]. The rays in the packet are synchronized to operate on the same B-KD tree node, which reduces the memory bandwidth. This multi-threading and packet-based approach performs very well because of the high coherence between adjacent rays. The memory bandwidth is reduced further by using dedicated first level caches to store B-KD tree nodes.

The implemented traversal algorithm for the B-KD tree is similar to that of standard KD trees [22]. The recursive traversal function traverses the scene in a traversal interval $I = [near, far]$ along the ray. We first test for early ray termination by determining if the current hit is before the *near* distance. We then intersect the ray with the four bounding planes defined by the node giving the two intersection intervals $I_{\{0,1\}}$ for the two leaf nodes (see Figure 4). A child that lies partially in front of the second one is traversed first, as it is more likely that the closest hitpoint is located there. Before this closer child (for instance child 1) is traversed two comparisons determine if its intersection interval I_1 overlaps the current traversal interval I. We recursively traverse the child if this is the case. The traversal interval is then updated to the intersection of I and I_1, which requires two min/max operations. If the other child overlaps the traversal interval it is stored onto a stack together with the intersection of I and I_0 as its traversal interval.

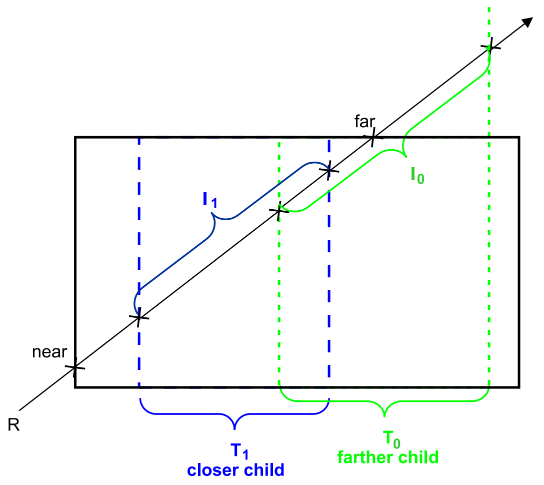

Figure 4: Ray Traversal: The ray is intersected with the four planes defined by the bounds of each child giving two intersection intervals $I_{\{0,1\}}$ along the ray. A child is traversed iff its intersection interval overlaps the traversal interval $I = [near, far]$ of the ray. The closer child is always traversed first to improve performance through early ray termination.

3.3 Geometry Unit (GU)

If the Traversal Processor reaches a leaf node, the Geometry Unit (GU) is responsible for sequentially intersecting the rays of a packet with the contained triangle geometry using the Möller-Trumbore algorithm [14], or to sequentially transform rays to the local coordinate space defined by a transformation node of the B-KD tree. This transformation requires no additional arithmetic units as they can be shared with the ones used for ray/triangle intersection. The Geometry Unit is pipelined and can perform one ray triangle intersection or one ray transformation every two clock cycles. Thus eight cycles are required to transform or intersect the four rays of a packet.

4 Shader Processor (SP)

At its core the DRPU architecture contains a general purpose Shader Processor (SP) similar to the SPUs used in the RPU architecture [35]. It supports random memory read and write operations as well as arbitrary address computations using integer arithmetic. However, the design has been optimized for algorithms with properties similar to those of ray tracing: generous thread-level parallelism, high data coherence between threads of nearby pixels, and a large number of short vector floating point operations. Via a special “trace” instruction it can recursively call the Ray Casting Units described in the previous section for efficient traversal of additional rays through the index structure. On a “trace” instruction the ray and a pointer to the spatial index structure are sent to the Traversal Processor (TP). The TP performs the traversal of the ray with that structure and writes results to special return registers. The SP may now continue operating on the thread-packet using the information provided by the TP.

Similar to current GPUs we use four component, single precision floating point or integer vectors as the basic data type in the core

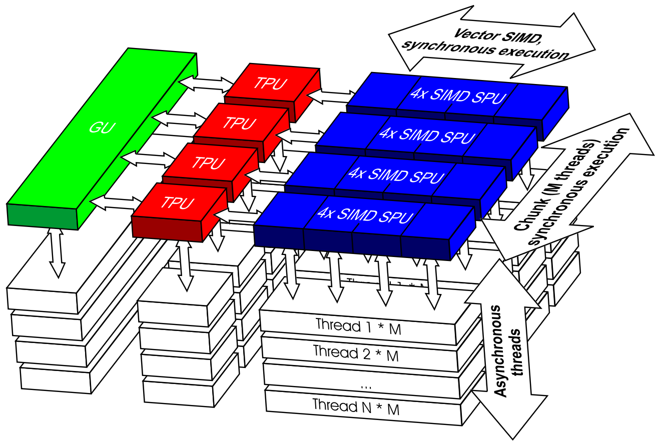

Figure 5: An abstract view of the implemented DRPU. Each of the four Shader Processing Units (SPUs) operates on four-component vectors as its basic data type, and all four SPUs operate synchronously in SIMD mode making up a 4-thread packet. A total of 32 packets are supported and executed asynchronously in the multi-threading hardware for a total of 128 supported hardware threads. Four Traversal Processing Units (TPUs) synchronously traverse packets of four rays through a B-KD tree using the Geometry Unit (GU) for ray/triangle intersection.

Shader Processing Unit (SPU) to exploit the available instruction level parallelism. This results in fewer memory requests of larger size, and significantly reduces the size of the shader programs compared to a scalar code. Again similar to GPUs, dual-issue instructions are supported to split the vector into two parts and perform different computations on them, or to pair an arithmetic instruction with a load or branch instruction.

We take advantage of the *thread parallelism* in ray tracing through a massively multi-threaded hardware design with 128 hardware threads supported in the implemented version of the DRPU.

5between threads as required. Multi-threading allows an increase in

The raw bandwidth requirement of the unmodified ray tracing algorithm is huge [20]. It can be reduced considerably by exploiting the high coherence between adjacent rays. To this end, four threads are packed into a packet and executed *synchronously in SIMD mode* in parallel by four SPUs in the hardware (see Figure 5). There are 32 of these four-thread packets supported. Because all threads in a packet execute the same instruction, identical memory requests are highly likely for coherent rays in the packet and can be combined. Using SIMD mode, these SPUs can share much of their infrastructure (e.g. instruction scheduling and caches), which reduces the hardware complexity. The current numbers of four threads per packet and 32 packets were chosen after detailed simulation as a good balance between hardware complexity and available memory bandwidth in the current hardware. An increase of the number of threads would yield higher performance, but a slightly sublinear relation to the required additional space makes 32 packets a good compromise. On the other hand, increasing the number of synchronous rays per packet to more than four could cause problems during Place and Route of the ASIC design. A synchronization circuit is required to synchronize between the single units and larger packets could cause this circuit to be far away from some of the units. Furthermore, very large packets would reduce the performance for incoherent computations such as highly triangulated scenes, because few rays would active during the computation.

In order to allow for complex control flow even in a SIMD environment the architecture supports conditional branching and full recursion using masked execution and a hardware-maintained register stack accessible through the register file. Unused parts of this stack can transparently be spilled out to main memory by the hardware to allow for deep recursions. Diverging branches of the threads in the packet are automatically handled sequentially by processing one control path and putting instruction pointer and activity mask of the second one onto a control stack. In the executed control path, an activity mask determines which threads take part at the computations. If the current control path reaches a return statement the next item of the control stack is executed.

Memory requests are a key issue with multi-core designs. It turns out that the synchronous execution of rays leads to many identical memory requests that can be packed and thus reduce bandwidth. This memory packing mechanism only performs the required number of memory requests for the packet of rays, e.g. if each ray of the packet wants to read data from the same address only one packed memory request is performed. Nevertheless incoherent packets are allowed and cause no additional overhead but do not see improvements either as four single requests are performed. All memory accesses go through small dedicated caches (see Figure 2) in order to further reduce external bandwidth and re-use data between different packets of threads. Cache hit rates are generally much higher than 90% in our test scenes which results on low external bandwidth requirements (see Table 2).

The main difference of the DRPU to the RPU design as described in [35], is the special Geometry Unit, that the DRPU uses for ray/triangle intersection based on shared triangle vertices. This unit is required to handle dynamic scenes as precomputing acceleration data to speed up a ray/triangle intersection in software on the SP is difficult as this computation requires matrix inversions and a second iteration over the dynamic geometry. Also this would greatly increase the size of the scene database, as no vertices could be shared, resulting in more memory traffic. The RPU design had only limited support for rigid-body motion, not for highly dynamic scenes as supported by the DRPU by using B-KD trees.

5 DRPU Implementation

5.1 FPGA Prototype

To accurately estimate the performance of our ASIC design, we implemented an FPGA version of the DRPU on a Xilinx Virtex-4 LX 160 FPGA [36] that is hosted on the Alpha Data ADM-XRC-4 PCI-board [1]. The FPGA has access to four 16-bit wide DDR memory chips used in parallel to make a 64-bit wide memory interface that can deliver a peak bandwidth of 1.0 GB/s at 66 MHz. The FPGA is connected via a 64 bit wide PCI bus to the host PC. The DMA capabilities of the PCI bridge are used to upload scene data (B-KD tree nodes, shader code, and all shader parameters) to DRAM and to download frame buffer contents to the application for storage or display via standard graphics APIs.

The hardware description of the entire DRPU prototype is about 8000 lines of ML [13] code using the HWML library for hardware description [33]. The specification is fully parameterizable, thus each of the design parameters, like packet size, number of threads, latencies, floating point accuracy, and caches, can be changed by adjusting a single configuration file. We adjusted the configuration to achieve the best possible performance with our FPGA by completely using the available logic resources.

Due to the limited size of the FPGA not all features of the DRPU architecture could be enabled for the prototype: integer operations are not included, which limits memory reads to offsets of precomputed addresses. Write support is limited to a single vector per shader (similar to GPUs). A fixed register stack of 16 entries is provided without automatic spilling of unused parts to memory. In order to take advantage of the available 18 bit multipliers on the Xilinx FPGA, a 24-bit floating point format was used. With a packet size of 4 and 32 packets (128 hardware threads total) the DRPU occupies about 99% of the logic cells, 165 of the 288 block memories (57%), and 58 of the 96 18-bit multipliers (60%) of the FPGA chip. These numbers show that we use the FPGA to its limits which sometimes causes problems with routing and overmapping. The design contains 113 floating point units, mostly in the SPUs, TPUs, and GU. The worst-case timing according to the Xilinx mapping tools is 55 MHz, but the DRPU runs at 66 MHz as implemented. At this clock speed the theoretical peak performance is 7.5 GFlops.

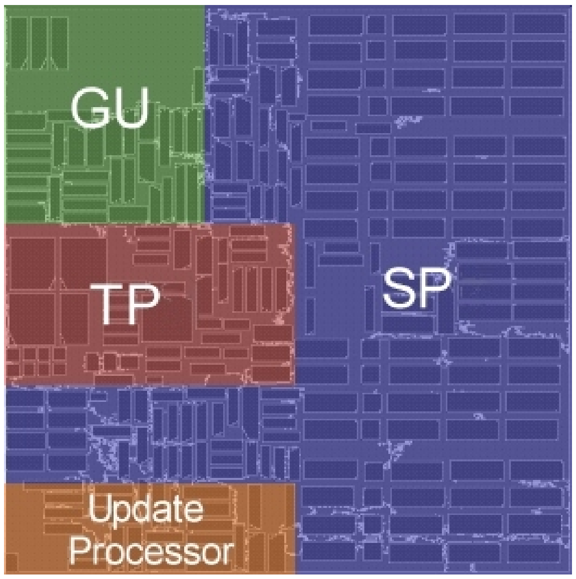

Figure 7: Plot of the DRPU ASIC shown with only four of the six levels of metal wiring so that the memories are visible. The Shader Processor (SP), Traversal Processor (TP), Geometry Unit (GU), and Update Processor are shaded, to show their die area and complexity. As we did not designed the external connection of the chip (PCI plus DRAM interfaces) pads are not included in the Figure.

5.2 ASIC Design

For the ASIC version of the DRPU we mapped the HWML-generated description to a set of standard cells in a 130nm CMOS process from UMC [24]. For on-chip RAM we used memories generated by an SRAM memory compiler from Virtual Silicon. Physical assembly and post layout timing was done using the Cadence SOC Encounter tools.

The ASIC version does not suffer from space limitations, thus we increased the floating point data path to full 32-bit single-precision width, including integer arithmetic and other features that had been disabled in the FPGA version. To easily estimate performance we configured the DRPU ASIC version in a similar way to the FPGA version with packet size 4 and 32 packets. This also results in 113 floating point units on the DRPU ASIC. To make a conservative performance extrapolation, we additionally increased the cache sizes and implemented four-way set associative caches for the SP (16 KBytes), TP (16 KBytes), and GU (16 KBytes). The total core size for this DRPU is 7mm x 7mm (49 mm^2) in the 130nm CMOS process. The post layout timing estimates for the current version are 161 MHz worst case (1.08V, 125°C) and 299 MHz typical case (1.25V, 25°C). These speed estimates are approximately 70% of the maximum possible speed of the on-chip memories generated with our memory compiler, which shows room for further improvements. A clock rate of 266 MHz should easily be achievable if the chip would be fabricated and would have a theoretical peak perfor-

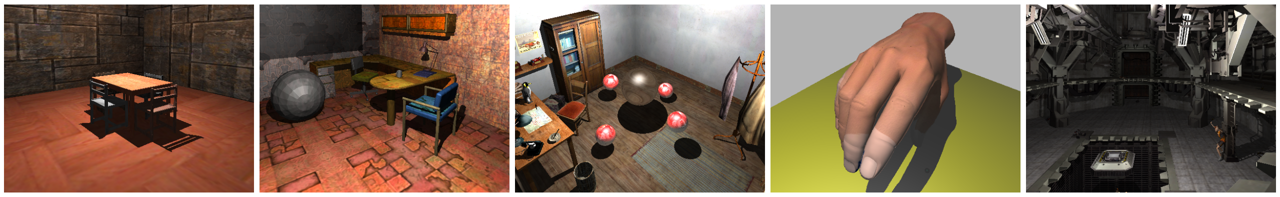

Figure 6: Some of the the scenes used for benchmarking the prototype: *Scene6* (0.5k triangles), *Office* (34k triangles), *Mafia Spheres* (20k triangles), *Hand* (17k triangles), and *Gael* (52k triangles). See Figure 1 for more benchmarking scenes.

mance of 30.0 GFlops.

A plot of the DRPU layout is shown in Figure 7 with the memories visible as the large blocks and some hardware units are labeled and shaded. In total there are approximately 9 million non-memory transistors in the DRPU (686k standard cells, 191k of them are flip flops) and approximately 2.57 MBit of on-chip RAM in the caches (0.6 MBit), register files (1.2 MBit), and other memory structures (0.77 MBit) that are implemented in 280 generated memory blocks.

6 Performance Evaluation

DRPU FPGA: The fully functional FPGA prototype, configured as described in Section 5.1, runs at 66 MHz with 1GB/s peak memory bandwidth between the on-board SDRAM and the on-chip caches. It turns out that half the peak memory bandwidth is sufficient for most of our test scenes, thus for measurements we scaled the available bandwidth down to only 0.5 GB/s using some test circuits. The performance of the DRPU FPGA is measured directly from the running hardware by counting the number of cycles required to update spatial index structures and to compute the image (see Table 2).

DRPU ASIC: The timing of the DRPU ASIC, configured as described in Section 5.2 is estimated from post layout timing analysis using Cadence SOC Encounter. Because the architecture is the same, and the ASIC clock rate is four times higher than for the FPGA, we can derive performance numbers for this ASIC version by scaling the FPGA framerates linearly. This is precise as long as the external memory bandwidth could also be scaled linearly to 2.1 GB/s. Because of the larger caches of the ASIC this performance estimate is quite conservative.

Todays high end rasterization graphics chips like the ATI R520 use a 90nm process with a 288mm^2 die. This is much larger than our DRPU ASIC version whose die is 49mm^2 large and uses a 130nm process. For this reason we estimate performance for two further ASIC versions with larger die size and with a 90nm process (see Table 3).

DRPU4 ASIC: First we maintain the process and put four copies of the basic DRPU ASIC on a single chip. We did no ASIC layout for this DRPU4 ASIC version, but it would fit on a 14mm x 14mm die (196 mm^2) at 130nm if one ignores the area required for connecting the four DRPU copies to main memory. If run at 266 MHz the 452 floating point units of the DRPU4 ASIC would provide a peak floating point performance of 120.0 Gflops. The performance could again be scaled up linearly if the chip would be connected to a DDR memory interface with 8.5 GB/s peak bandwidth, which can be implemented quite feasibly with two 64-bit wide DDR2 memory interfaces clocked at effective 532 MHz.

DRPU8 ASIC: Next we extrapolate performance levels that could be achieved with the DRPU design by going from our 130nm process to a 90nm process. Because we don't have access to this process, we cannot provide precise timing results from Cadence SOC Encounter, but extrapolations using constant field scaling are reasonably accurate [31]. If one scales the dimensions of a process by s using constant field scaling, then the frequency scales by a factor of $1/s$. If we extrapolate from our 130nm design to a 90nm process, s is 0.69 and we get a maximal operating frequency of 299 $MHz/0.69 = 433\ MHz$ for the DRPU. Thus we consider a 90nm version running at 400 MHz. Feature size decreases by s, thus the DRPU ASIC has a die size of 4.83mm x 4.83mm = 23.3mm^2 in the 90nm process, and we can instantiate eight copies on a 186.6mm^2 die. To provide enough memory bandwidth we would need to connect this DRPU8 ASIC to a 25.6 GB/s memory interface. External memory interfaces at that speed are difficult to implement, but realistic if looking to current high end GPUs with external bandwidths of more than 40 GB/s. Again the memory interface and connection to the Rendering Units would consume additional die area. The DRPU8 ASIC would have an additional 3 times speedup over the DRPU4 ASIC, because of higher frequency and twice the number of computational units. The 904 floating point units would provide a peak floating point performance of 361 GFlops, which is very close to the peak floating point performance of todays GPUs. Because of the high rendering performance, a high speed PCI Express connection would be required to download the rendered pixels for display.

Table 1 gives an overview of frequency, peak floating point performance, and die characteristics for the FPGA version, the different ASIC design versions, the Pentium 4 chip and the Cell processor used for speed comparison.

We have not done detailed power analysis or estimation of any of the ASIC versions, but we would expect power to be high, due to the large number of floating point units and the computational requirements of the rendering process. In this dimension we expect no particular improvements over existing GPUs which also exhibit high power consumption.

To show the possible performance, we have chosen a number of benchmark scenes (see Figure 1 and Figure 6) that cover a large fraction of possible scene characteristics. The scenes range from very simple ones like the Shirley6 and Office scenes, to complex ones (Conference) and the Gael level from Unreal Tournament 2004. The Mafia Spheres scene, shows a room containing four reflecting and one refracting sphere, to show secondary ray tracing

	OpenRT P4	PS3-Cell	DRPU FPGA	DRPU ASIC	DRPU4 ASIC	DRPU8 ASIC
Freq [MHz]	2,667.0	3,200.0	66.0	266.0	266.0	400.0
GFlops	10.6	256.0	7.5	30.0	120.2	361.6
process [nm]	130	90	90	130	130	90
die size [mm^2]	145.0	221.0	-	49.0	196.0	186.6
bandwidth [GB/s]	8.5	25.0	0.5	2.1	8.5	25.6

Table 1: Comparison of the different hardware architectures: the OpenRT software implementation running on a Pentium 4, the Cell implementation, the DRPU FPGA implementation, the DRPU and DRPU4 ASIC implementations on a 130nm process, and the extrapolation of the DRPU8 ASIC to a 90nm process.

Scene	triangles	objects	#rays	cycles update	cycles render	FPGA framerate	cache hitrates TP	cache hitrates GU	cache hitrates SP	DRAM bandwidth
Shirley6	0.5k	1	1.5M	-	14M	4.7 fps	98.6%	99.2%	85.7%	113 MB/s
Conference	282k	52	1.5M	-	39M	1.7 fps	81.3%	85.1%	89.6%	164 MB/s
Office	34k	1	1.5M	-	18M	3.6 fps	91.5%	93.7%	88.0%	103 MB/s
Mafia Room	15k	1	1.5M	-	24M	2.8 fps	91.4%	96.3%	67.2%	186 MB/s
Mafia Spheres	20k	6	1.6M	-	36M	1.8 fps	88.7%	96.1%	59.8%	210 MB/s
Hand	17k	2	1.3M	118k	13M	5.0 fps	91.8%	97.9%	75.3%	126 MB/s
Skeleton	16k	2	1.3M	113k	11M	5.9 fps	89.8%	97.5%	96.3%	73 MB/s
Helix	78k	2	1.5M	602k	18M	3.5 fps	80.0%	93.2%	87.2%	145 MB/s
Gael	52k	1	1.5M	-	34M	1.9 fps	87.7%	91.4%	72.1%	188 MB/s
DynGael	85k	4	1.5M	121k	33M	2.0 fps	86.1%	91.6%	88.0%	154 MB/s

Table 2: Performance statistics of the DRPU FPGA prototype clocked at 66 MHz. For several scenes, the complexity in number of triangles, instantiated objects, and number of rays shot per image at 1024x768 resolution are shown. Further, the table shows the number of cycles required for updating the B-KD tree, for rendering the image, and the resulting framerate. Cache hitrates are shown for the TP, GU, and SP cache. The low resulting external memory bandwidth is presented, showing the scalability of the approach. Phong shading is used including textures and *shadows*. The cycles required to read back the framebuffer contents for display are not included (but would be below 1% for most scenes). See Figures 1 and 6 for images of the scenes.

Scene	triangles	objects	#rays	DRPU FPGA	DRPU ASIC	DRPU4 ASIC	DRPU8 ASIC
Shirley6	0.5k	1	1.5M	4.7 fps	18.8 fps	75.2 fps	225.6 fps
Conference	282k	52	1.5M	1.7 fps	6.7 fps	27.0 fps	81.2 fps
Office	34k	1	1.5M	3.6 fps	14.4 fps	57.6 fps	172.8 fps
Mafia Room	15k	1	1.5M	2.8 fps	11.2 fps	44.8 fps	134.4 fps
Mafia Spheres	20k	6	1.6M	1.8 fps	7.2 fps	28.8 fps	86.4 fps
Hand	17k	2	1.3M	5.0 fps	20.0 fps	80.0 fps	240.0 fps
Skeleton	16k	2	1.3M	5.9 fps	23.6 fps	94.4 fps	283.2 fps
Helix	78k	2	1.5M	3.5 fps	14.0 fps	56.0 fps	168.0 fps
Gael	52k	1	1.5M	1.9 fps	7.6 fps	30.4 fps	91.2 fps
DynGael	85k	4	1.5M	2.0 fps	8.0 fps	32.0 fps	96.0 fps

Table 3: Estimated performance of the DRPU versions for a number of benchmark scenes of varying complexity. We provide the number of cycles required for updating of the B-KD tree and rendering of the images at 1024x768 resolution *with shadows*, Phong shading, and textures. Frames per second are directly computed from the number of cycles required for the computation. The cycles required to read back the framebuffer contents for display are not included (but would be below 1% for most scenes). See Figures 1 and 6 for images of the scenes.

Scene [fps]	OpenRT	Cell	DRPU FPGA	DRPU ASIC	DRPU4 ASIC	DRPU8 ASIC
Shirley6	3.2	180.0	5.0	20.0	80.0	240.0
Office	2.6	n/a	4.1	16.4	65.6	196.8
Conference	2.0	60.0	3.4	13.6	54.4	163.2
Gael	2.0	n/a	3.8	15.2	60.8	182.4

Table 4: Performance comparison of the OpenRT software implementation running on a Pentium 4 with 2.66 GHz, the Cell ray tracer, the DRPU FPGA running at 66 MHz, post-layout estimates for the DRPU ASIC, and estimates for the DRPU4 ASIC running at 266 MHz and the DRPU8 ASIC running at 400 MHz. All performance numbers are for 1024x768 resolution with phong shading including bilinear texturing, vertex normal interpolation, and a single light source *without shadows*.

effects. Some Poser [18] animations (Hand, Skeleton, and Helix) show the support for dynamic scenes. The vertex positions and normals are precomputed by Poser, and uploaded via DMA for each frame. The DynGael scene, shows the combination of the static Gael level, with two dynamic skeleton instances.

We use a subset of these scenes for speed comparisons, see Table 4. A comparison of the performance of the FPGA prototype and the three ASIC versions against the OpenRT software ray tracer running on an Intel Pentium-4 at 2.66 GHz [4] and a Cell implementation of ray tracing [3] are performed. The results show that the FPGA version outperforms the software implementation by 40% to 70% even though clocked at a 40 times lower frequency. The DRPU8 ASIC version would outperform the software ray tracer by a factor of up to 75. A comparison to a Cell implementation of ray tracing shows up to 2.5 times higher performance, despite the hardware complexity being similar (see Table 1), and the DRPU8 ASIC performing much more complex shading (including textures). This shows the efficiency of the DRPU architecture compared to general purpose designs.

For the full set of test scenes, detailed statistics in Table 2 and performance extrapolations in Table 3 are provided. The statistics include the complexity of the scene, number of object instances, and number of rays shot for computation. The scenes are rendered with a realistically complex shader with more than 90 assembly instructions to perform: bilinear texture lookup, diffuse term, specular term, light fall-off, vertex normal interpolation, vertex color interpolation, and pixel accurate shadows. Table 2 further shows the exact number of cycles required to update the B-KD trees, rendering, and the resulting frame rate of the FPGA. The cache hit rates of the direct mapped FPGA caches are an indicator for the coherence of the computations, but typically are much higher than 90%. The hit-rates drop down especially for higher resolution textures that are accessed by the SP. As the ASIC versions implement four-way set associative caches with twice the size as the FPGA, much higher hit-rates are expected, which would reduce external bandwidth even more.

The performance extrapolations of Table 3 show performance for the DRPU FPGA, and all DRPU ASIC versions. If comparing these performance values against Table 4, the numbers for the Shirley6 and Office scene are surprisingly only slightly lower despite containing shadows. This is because traversal and shading can be performed in parallel and for these two simple scenes the Ray Casting Units can trace the shadow ray at the same time as the SP performs shading. For the test scenes, the estimated performance of the DRPU8 ASIC is between 80 and 280 frames per second. The performance mainly depends on the cost of the rays, which increases with higher number of visible geometry elements, and the total number of rays shot. Thus the performance of the Mafia Spheres scene is lower than the Mafia Scene, because more triangles are visible and more rays need to be shot due to the refraction and reflection effect.

The Gael level renders with more than 90 frames per second at 1024x768 resolution even with two animated Skeleton instances. This is sufficient for game play and would leave much room for improved image quality. For instance, it would be possible to improve filtering of edges and shadows by using adaptive oversampling techniques. These techniques take an additional pass over the generated image to find regions where more rays could effectively improve image quality.

The DRPU hardware architecture can render even highly dynamic scenes efficiently, as shown by the results of the Hand, Skeleton, and Helix animations. For these dynamic test scenes the number of cycles required to update the B-KD tree is about two orders of magnitude below the render cycles, and rendering these animations causes little overhead.

7 Conclusions

This paper presents ASIC implementations of the programmable DRPU architecture for efficient high performance ray tracing of dynamic scenes. The DRPU contains a Shading Processor implemented as a four-element vector floating point processor core with support both for synchronous SIMD execution of packets of threads and multithreading. It also contains custom hardware for ray/triangle intersection and for traversing the B-KD tree which is required for efficient ray tracing of dynamic scenes.

The FPGA prototype is fully working and makes for convincing demonstrations of the power of this technique. We hope to fabricate at least the single-DRPU ASIC to demonstrate the full potential of this architecture. Measurements on the implemented FPGA prototype, and timings based on a 130nm ASIC design indicate that performance levels sufficient for game play are achievable, especially if it is possible to use a high end 90nm ASIC technology. A DRPU would also offer much higher quality of image and realism due to the use of recursive ray tracing rather than rasterization.

References

[1] Alpha-Data. ADM-XRC-II. *http://www.alphadata.uk.co*, 2003.

[2] Arthur Appel. Some Techniques for Shading Machine Renderings of Solids. *SJCC*, pages 27–45, 1968.

[3] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich. Ray Tracing on the CELL Processor. In *IEEE Symposium on Interactive Ray Tracing*, 2006.

[4] Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek. The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium*, pages 23–31, Darmstadt, Germany, 2003. EUROGRAPHICS Association.

[5] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM Press.

[6] Stuart A. Green. Parallel processing for computer graphics. *MIT Press*, pages 62–73, 1991.

[7] Stuart A. Green and Derek J. Paddon. A highly flexible multiprocessor solution for ray tracing. *The Visual Computer*, 6(2):62–73, 1990.

[8] M. Porrmann H. Kalte and U. Rückert. Using a dynamically reconfigurable system to accelerate octree based 3D graphics. Technical report, System and Circuit Technology, University of Paderborn, 2000.

[9] D. Hall. The AR350: Today's ray trace rendering processor. In *Proceedings of the EUROGRAPHICS/SIGGRAPH workshop on Graphics Hardware - Hot 3D Session*, 2001.

[10] Hiroaki Kobayashi, Kenichi Suzuki, Kentaro Sano, and Nobuyuki Oba. Interactive Ray-Tracing on the 3DCGiRAM Architecture. In *Proceedings of ACM/IEEE MICRO-35*, 2002.

[11] Jonas Lext and Tomas Akenine-Möller. Towards Rapid Reconstruction for Animated Ray Tracing. In *Eurographics 2001 – Short Presentations*, pages 311–318, 2001.

[12] Tony T.Y. Lin and Mel Slater. Stochastic Ray Tracing Using SIMD Processor Arrays. *The Visual Computer*, pages 187–199, 1991.

[13] Robin Milner, Mads Tofte, and Robert Harper. The Definition of Standard ML, 1990.

[14] Tomas Möller and Ben Trumbore. Fast, minimum storage ray triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[15] Jean-Christophe Nebel. A Mixed Dataflow Algorithm for Ray Tracing on the CRAY T3E. In *Third European CRAY-SGI MPP Workshop*, September 1997.

[16] 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.

[17] Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The VolumePro real-time ray-casting system. *Computer Graphics*, 33, 1999.

[18] Poser. Poser Web Page. *http://www.e-frontier.com*, 2006.

[19] Timothy J. Purcell. *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, 2004.

[20] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 27–36, 2002.

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

[22] K. R. Subramanian. *A Search Structure based on K-d Trees for Efficient Ray Tracing*. PhD thesis, The University of Texas at Austin, December 1990.

[23] Tomas Akenine-Möller Thomas Larsson. Strategies for bounding volume hierarchy updates for ray tracing of deformable models. Technical report, February 2003.

[24] United Microelectronics Corporation. http://www.umc.com, 2005.

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

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

[27] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Interactive Global Illumination in Complex and Highly Occluded Environments. In Per H Christensen and Daniel Cohen-Or, editors, *Proceedings of the 2003 EUROGRAPHICS Symposium on Rendering*, pages 74–81, Leuven, Belgium, June 2003.

[28] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies (revised version). *Technical Report, SCI Institute, University of Utah, No UUSCI-2006-023*, 2006.

[29] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM SIGGRAPH 2006*, 2006.

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

[31] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison Wesley, 2005.

[32] Turner Whitted. An Improved Illumination Model for Shaded Display. *CACM*, 23(6):343–349, June 1980.

[33] Sven Woop, Erik Brunvand, and Philipp Slusallek. HWML: RTL/Structural Hardware Description using ML. Technical report, Computer Graphics Lab, Saarland University, 2006.

[34] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware*, 2006.

[35] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. In *SIGGRAPH 2005 Conference Proceedings*, pages 434 – 444, 2005.

[36] Xilinx. Virtex-II. *http://www.xilinx.com*, 2003.