HART: A Hybrid Architecture for Ray Tracing Animated Scenes

Jae-Ho Nah, Jin-Woo Kim, Junho Park, Won-Jong Lee, Jeong-Soo Park, Seok-Yoon Jung, Woo-Chan Park, Dinesh Manocha, and Tack-Don Han

Abstract—We present a hybrid architecture, inspired by asynchronous BVH construction [1], for ray tracing animated scenes. Our hybrid architecture utilizes heterogeneous hardware resources: dedicated ray-tracing hardware for BVH updates and ray traversal and a CPU for BVH reconstruction. We also present a traversal scheme using a primitive's axis-aligned bounding box (PrimAABB). This scheme reduces ray-primitive intersection tests by reusing existing BVH traversal units and the primAABB data for tree updates; it enables the use of shallow trees to reduce tree build times, tree sizes, and bus bandwidth requirements. Furthermore, we present a cache scheme that exploits consecutive memory access by reusing data in an L1 cache block. We perform cycle-accurate simulations to verify our architecture, and the simulation results indicate that the proposed architecture can achieve real-time Whitted ray tracing animated scenes at 1920×1200 resolution. This result comes from our high-performance hardware architecture and minimized resource requirements for tree updates.

Index Terms—Ray tracing, bounding volume hierarchy, dynamic scene, graphics hardware

1 Introduction

Recently, a great deal of research has been conducted to achieve ray tracing dynamic scenes at interactive rates [2]. In dynamic scenes, objects can be moved, added or deleted from a scene, or animated with topological changes. Because most ray-tracing systems are based on acceleration data structures, such as kd-trees, bounding volume hierarchies (BVHs), and grids, these acceleration data structures should be effectively updated for dynamic scenes. Many researchers have exploited CPUs [1], [3]–[11], GPUs [12]–[16], MIC (many integrated core) [11], [17], or dedicated ray-tracing hardware [18]–[20] to achieve this goal.

However, most current real-time rendering engines (e.g. game engines) use techniques based on rasterization instead of ray tracing. This means that current ray-tracing systems still do not provide sufficient performance for the real-time rendering of dynamic scenes on commodity hardware. To achieve ray-tracing in dynamic scenes at real-time rates, there are two requirements: to get high-quality effects, the ray traversal performance must be high; and there must be fast acceleration-data-structure updates that do not degrade the tree quality.

To achieve these two goals, we present a hybrid ray-tracing architecture based on the BVH. In this architecture, dedicated ray-tracing hardware performs traversal and ray-triangle intersection tests because these two operations tend to be the main bottlenecks in ray tracing. In order to deal with dynamic scenes, we extend CPU-based asynchronous BVH construction schemes [1], [7]; tree construction is performed using a CPU, and bounding volume (BV) refitting is performed by geometry and tree update (GTU) units, as part of ray-tracing hardware. This approach greatly reduces the tree update cost because expensive BVH construction does not need to be performed during each frame.

When we utilize multiple hardware resources, the throughput of each hardware component and the communication cost between the hardware components determine overall performance. For our system, we present a traversal scheme using primitive's axis-aligned bounding boxes (primAABBs) with shallow trees. In this scheme, the traversal unit in the ray-tracing hardware performs both BVH traversal and ray-primAABB intersection tests. When this scheme is used with shallow trees, it reduces both tree build times and tree sizes by up to 44% without significant performance degradation.

We also present a cache scheme for our traversal unit; we maintain two consecutive sets of shape data (node and primAABB) in an L1 cache block to reuse the data in the next iteration. This cache-data reuse scheme reduces cache misses caused by eviction and increases rendering performance up to 21%.

We verify the performance of our architecture using a cycle-accurate simulator. According to the simulation results, our architecture could achieve a significantly higher performance in ray-tracing dynamic scenes than other systems [17], [21], [22]. Conse-

J.-H. Nah, J.-W. Kim, J. Park, and T.-D. Han (corresponding author) are with Yonsei University, E-mail: {jhnah, jwkim, bluedawn, hantack}@msl.yonsei.ac.kr (part of the work was done when J.-H. Nah was visiting UNC Chapel Hill)
W.-J. Lee, J.-S. Park, and S.-Y. Jung are with Samsung Electronics, E-mail: {joe.w.lee, js1980.park, seokyoon.jung}@samsung.com
W.-C. Park is with Sejong University, E-mail: pwchan@sejong.ac.kr
D. Manocha is with the University of North Carolina at Chapel Hill, E-mail: dm@cs.unc.edu

quently, our architecture can perform ray-tracing dynamic scenes with full Whitted effects (e.g., specular reflection and hard shadows) at real-time rates. This result comes from the efficient use of each hardware component and from the maintenance of BVH quality. In static scenes, our architecture performs comparably to other ray-tracing hardware architectures designed for static scenes [23].

2 Previous Work

In this section, we summarize prior work on asynchronous BVH construction. Next, we introduce related work, including ray-tracing hardware architectures, CPU-GPU hybrid ray tracers, and primitive culling algorithms.

2.1 Asynchronous BVH Construction

There are two representative BVH update methods: BV refitting and rebuilding. First, BV refitting [3], [4] quickly updates the BVs without changing the topology of the BVHs. However, this method degrades the tree quality due to overlap between BVs. In contrast, BVH rebuilding methods [5] reconstruct the BVH from scratch during each frame. This method creates high-quality trees but takes longer than BV refitting.

The rebuild heuristic [3] detects tree-quality degradation after BV refitting to determine when the BVH needs to be rebuilt, but this method may cause a disruptive pause while the BVH is being rebuilt [7]. Selective restructuring [6] continuously reconstructs subtrees instead of rebuilding the entire tree at certain points, which prevents that disruptive pause. However, this method proceeds in a serial manner, so it is not suitable for our parallel architecture. Lauterbach et al. [24] proposed an oriented bounding box (OBB)-based refitting process for collision detection. This OBB-based process, however, cannot be directly used for our AABB-based ray-tracing system. Finally, a tree-rotation algorithm [9] performs additional tree-rotation operations after BV refitting to reduce tree-quality degradation.

Asynchronous BVH construction [1], [7], the base algorithm used in our architecture, asynchronously executes BV refitting and BVH rebuilding on a multi-core CPU. While a new BVH is built on specific threads, the remaining threads perform BV refitting and rendering during each frame. This approach takes the best of both methods: it prevents BVH quality degradation from BV refitting while maintaining frame rates, unlike the rebuild heuristic.

2.2 Dedicated Ray-Tracing Hardware Architecture

We classify the studies on dedicated ray-tracing hardware architectures into two categories: SIMD (single instruction, multiple data) approaches and MIMD (multiple instructions, multiple data) approaches. SaarCOR [25], RPU [26], and D-RPU [19] traverse four rays together by exploiting packet tracing and the four-wide SIMD architecture. These approaches are suitable for coherent rays, but they are inefficient for incoherent rays. To increase SIMD efficiency, StreamRay [27] filters active rays in a large packet.

In single-ray-based approaches, each ray is treated as an independent thread. Thus, these architectures have higher hardware utilization than packet-based SIMD approaches in the case of incoherent sets of rays. TRaX [28] and MIMD TM [29] distribute each ray to light-weight programmable cores. In contrast, the T&I engine [23] includes fixed pipelines for traversal and intersection operations. This architecture consists of traversal units using an ordered depth-first layout and three-phase intersection units. These two types of units commonly include a ray accumulation buffer for latency hiding. Aila and Karras [30] proposed a GPU-based single-instruction, multiple-threads (SIMT) architecture using treelets and a stack top cache to minimize memory traffic when tracing incoherent rays. SGRT [31], [32] is a mobile ray-tracing hardware architecture designed for static scenes. It combines dedicated T&I units and SRPs (Samsung reconfigurable processors). RayCore [33] is another mobile ray-tracing hardware architecture based on unified MIMD T&I units.

Many other ray tracing hardware architectures have been proposed for dynamic scenes. SaarCOR [25] includes a transformation unit for ray transformation. This hardware architecture does not update a kd-tree, so it is limited to piecewise rigid motion [18]. D-RPU [19] supports skinning animation through the use of a BKD-tree update unit [19]. This approach is similar to BV refitting [4] and is prone to problems related to tree-quality degradation [34]. In contrast, our architecture based on asynchronous BVH construction can maintain tree quality. Finally, Doyle et al. [20] proposed a hardware architecture for binned SAH BVH construction.

2.3 CPU-GPU Hybrid Ray-Tracing System

Some researchers have tried to utilize both CPUs and GPUs in a cooperative way to render dynamic scenes. Budge et al. [35] combined CPU tree construction and GPU ray tracing using CUDA. Nah et al. [36] implemented an OpenGL ES-based ray tracer for mobile devices, which assigned kd-tree construction and ray tree management to CPUs and ray traversal and shading to GPUs. The brigade renderer [37] is a rendering engine for path tracing dynamic scenes that exploits CPUs for game logic and BVH maintenance and GPUs for rendering.

Other researchers have focused on CPU-GPU hybrid path tracing static scenes. Budge et al. [38] distributed the tasks to CPUs and GPUs via careful scheduling. Combinatorial bidirectional path-tracing

[39] utilizes GPUs to link segments of camera and light paths and utilizes CPUs to avoid the limitations of pure GPU implementations. LuxRays [40] supports multiple OpenCL devices, so CPUs, GPUs, or both can be used for ray tracing. Unlike these studies, we focus on ray tracing dynamic scenes rather than static scenes.

2.4 Primitive Culling Algorithms for Ray Tracing

To reduce the computational costs of ray-primitive intersection tests, a few CPU-based primitive culling algorithms have been proposed. Vertex culling [41] substitutes a cheap ray-frustum test for unnecessary ray-triangle intersection tests. The ray box cull [42] creates a transient AABB using a ray's t interval in a grid cell and primAABB. Nah et al. [43] extended the ray box cull into kd-trees. These methods [41], [43] can be useful for ray tracing dynamic scenes when they are combined with shallow tree structures.

3 Proposed Architecture

We start this section by describing the overall system architecture and design decisions. We then introduce the traversal scheme using ray-primAABB intersection tests. Next, we describe the hardware components of our system in detail.

3.1 System Organization

Figure 1 illustrates the organization of our proposed system. This system consists of CPUs, ray tracing acceleration units, and programmable shaders. The goal of the proposed system is to utilize heterogeneous hardware resources for fast ray tracing dynamic scenes.

We chose asynchronous BVH construction [1] [7] for our system because we can easily distribute the BVH update process to a CPU and ray-tracing hardware. In our system, a CPU performs scene management and BVH construction. Because modern CPUs have multi-level cache hierarchies, surface area heuristic (SAH)-based tree construction [4] requiring random memory access fits well in these hierarchies. On the other hand, geometry and tree update (GTU) units in the dedicated hardware perform key-frame animation, BV refitting, and the computation of triangle data in [44] (triAccel) because these need to be performed during each frame. The input key-frame geometry data are transferred from the CPU to a memory in ray-tracing hardware via a PCI Express bus in advance, and they are used in the GTU unit. The computed data from the GTU unit are used for traversal and intersection (T&I) operations in ray tracing. The reason we chose a fixed hardware unit for this process is its high performance per area.

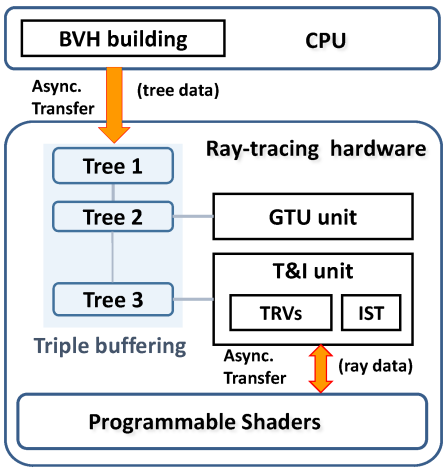


Fig. 1. Overall system architecture.

T&I units are comprised of fixed pipelines for high performance per area as well because T&I operations can dominate the computation of ray tracing. We used a single-ray-based approach rather than a ray-packet-based SIMD approach for efficient processing of incoherent rays. Additionally, efficient multi-threading is performed with a ray accumulation buffer [23] in each traversal and intersection unit. This buffer is used to prevent pipeline stalls by storing rays that induce a cache miss, and it permits the efficient concurrent processing of multiple rays in deep pipeline stages.

The traversal unit (TRV) performs both the BVH traversal and ray-primAABB intersection tests in Section 3.2. In contrast to the prior T&I engine [23], the TRV unit is optimized for BVHs rather than kd-trees.

We limited the primitive type to a triangle for simple configuration. The ray-triangle intersection unit (IST) is based on Wald's algorithm [44]. According to [23], this algorithm has the lowest cost of all ray-triangle intersection algorithms for hit triangles. Because of the increased possibility that the ray could hit the triangles in a leaf node after filtering of the ray-primAABB scheme, Wald's algorithm is a good choice for our architecture. Additionally, precomputation-based algorithms such as Wald's algorithm can be used to design an effective H/W intersection unit; consecutive memory access to precomputed data can simplify cache configuration and increase pipeline utilization compared to the Möller-Trumbore algorithm [45], which requires one index and three vertices. Although the precomputed triAccel data of Wald's algorithm increase memory footprints (40 bytes per triangle), we believe that the overhead is not high for scenes of moderate complexity.

Programmable shaders perform ray generation and shading to support various effects. We assume that these shaders are similar to unified shaders in commodity GPUs, and that ray data transmission between the T&I units and the programmable shaders use small FIFO buffers to reduce memory traffic, as with [23], [32]. We will only focus on BVH updates and

ray traversal in this paper; detailed descriptions of the ray generation and shading kernel are included in the SGRT paper [32].

3.2 Traversal Scheme using PrimAABBs

The proposed system in Section 3.1 has the following problems. First, asynchronous BVH construction on CPUs and our ray-tracing hardware require triple buffering of tree data (Figure 1): data transferred from CPUs, data for the GTU unit, and data for the T&I unit. Second, the BVH is sometimes too outdated by the time its construction is finished [7], and this situation occurs when the BVH construction time is long. Third, a big tree with a large number of nodes would create a bottleneck due to the limited bus bandwidth between the CPU and the ray-tracing hardware.

We deal with these issues by reusing the primAABBs for traversal. Originally, primAABBs are used for BVH building and BV refitting. We maintain primAABBs data after BV refitting and reuse them for traversal. When a ray reaches a leaf node, we perform a ray-primAABB test using the existing traversal unit before sending the ray to the intersection (IST) unit. This method substitutes most of the expensive ray-primitive intersection tests with ray-AABB intersection tests. For example, a ray-triangle intersection test requires 11–29 multiplications and 1–2 reciprocal instructions [23], but a ray-AABB intersection test only require 6 multiplications. We will explain the detailed traversal hardware architecture using this scheme in Section 3.4.

In combination with shallow trees, our traversal scheme using primAABBs solves all three of the problems listed above (Figure 2). Shallow trees with large leaf nodes have small memory footprints and also require less build time. However, these large leaf nodes incur more ray-primitive intersection tests. Our proposed scheme prevents traversal cost increases from the use of shallow trees as other culling methods do [41], [43]. However, our reuse of the existing traversal unit and primAABB data means that the additional culling stages required by other culling methods are unnecessary.

This method also permits an effective data layout. Wald's intersection algorithm [44] requires 36 bytes per triangle in a triAccel data, and a reordered triangle in a BVH leaf node requires its original triangle index for shading (4 bytes). If the precomputed triangle data (40 bytes) and a primAABB (24 bytes) are combined, 32-byte alignment can be made available (Figure 3) without padding. This configuration increases cache efficiency. If a ray passes the ray-primAABB test, we transfer the 8-byte triAccel data to the IST unit.

3.3 Geometry and Tree Update Unit

The geometry and tree update (GTU) unit computes BVHs, primAABBs, and triAccel data during each

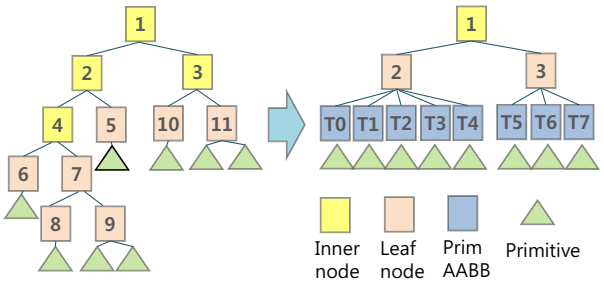


Fig. 2. An example of a shallow tree configuration using primAABBs: Our scheme reduces the tree depth and adds the primAABBs of each primitive to the tree. Each leaf node in the right figure points to primAABBs instead of actual primitives.

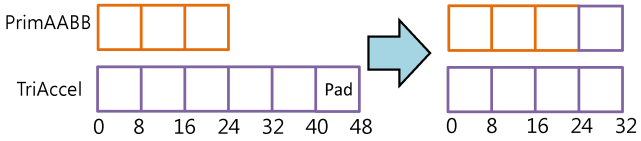


Fig. 3. 32-byte alignment by combining a primAABB with the precomputed triangle data (triAccel).

frame for animated scenes. This unit is organized into five pipeline stages as illustrated in Figure 4. Vertex and index fetch units read the triangle index and vertex data, an interpolation unit performs key-frame animation, and the AABB/triAccel calculation unit calculates each triangle's AABB and triAccel data. These three units were designed as a half-pipelined architecture for reduced hardware requirements. Finally, a BV refit unit performs BV refitting [4], and this unit is fully pipelined.

The index fetch unit fetches the index of three vertices in a triangle from the memory. The order of the triangles used in this unit corresponds to the triangle order in the leaf nodes of the BVH. In other words, all triangles in a leaf node are stored consecutively. This order removes triangle list fetching upon ray traversal to simplify the design of the T&I unit. Each index fetch unit has two index buffers for the concurrent processing of the index fetch unit and the vertex fetch unit.

The vertex fetch unit reads vertices from the external memory. Each vertex fetch unit has a 32-entry FIFO buffer to hide memory latency. If the buffer is not full, another thread can generate a memory request without a pipeline stall. The size of the memory request is 32 bytes. This policy reduces memory requests by using spatial locality when three vertices are adjacently stored in the memory.

The vertex interpolation unit calculates the interpolated vertices by using the two key frames. An interpolation requires nine multiplications and nine additions for three axes and three vertices. Each ver-

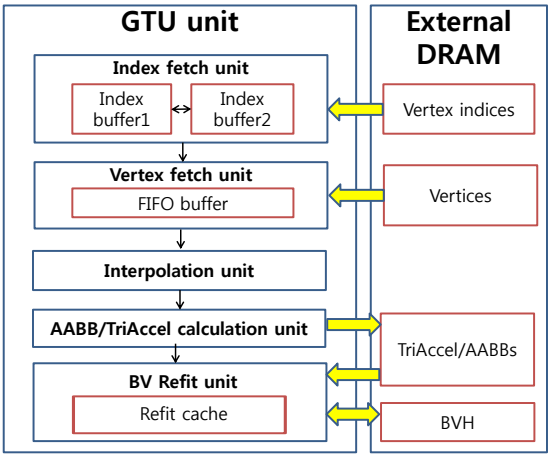


Fig. 4. The architecture of the geometry and tree update (GTU) unit.

tex interpolation unit consists of five multipliers and five adders. Thus, the throughput of the unit is 0.5 triangles per cycle.

The AABB/triAccel calculation unit calculates primAABBs and triAccel data using the interpolated vertices. Each AABB/triAccel calculation unit consists of 10 adders, 12 multipliers, and one reciprocal unit.

After the primAABBs are calculated, the BV refit unit performs BV refitting using the primAABBs. We used the breadth-first tree layout for BV refitting for easy parallelization. If the size of the node cache in the T&I units is 64 bytes, the breadth-first layout stores two child nodes in a cache block similar to the layout in [46]. Therefore, both layouts have the same cache efficiency in a 64-byte cache block. Each BV refit unit has a cache including node data and primAABB data; we also prefetch data into this cache to hide memory latency. The output data are transferred to the memory using the write-through policy.

The GTU procedure can be parallelized for large dynamic scenes. In this case, the geometry-update part (index fetch, vertex fetch, interpolation, and AABB/TriAccel calculation units) and the tree-update part (a BV refit unit) are separately operated. In the geometry update part, the number of triangles assigned to each parallel unit is the total number of triangles divided by the number of parallel units. In contrast, BV refit units exploit level-by-level parallelization in a bottom-up update manner [24]; at each level, the number of nodes assigned to each parallel unit is the number of nodes at the level divided by the number of parallel units. Only if all BV refit units finish the current updates, further BV updates at the upper levels of the tree are started.

3.4 Traversal and Intersection Unit

The traversal and intersection (T&I) unit (Figure 5) consists of one ray dispatcher (RD), 16 traversal (TRV) units, one TRV L2 cache, and one intersection (IST) unit. The RD gets rays from the programmable shaders and dispatches the rays to the TRV units. The RD also calculates the inverse direction vector for TRV units. Ray tracing is basically "embarrassingly parallel," and the parallelization of ray traversal can be easily achieved as with [23]. In other words, if enough ray threads are supplied to T&I units, the RD in each T&I unit can perform traversals using multiple TRV units.

The TRV units perform both BVH traversal and ray-primAABB intersection tests. Each TRV unit includes a ray-AABB intersection routine, stack memory, a ray accumulation buffer for latency hiding [23], and an L1 cache. The ray-AABB intersection calculation part is fully pipelined and it consists of six floating-point adders, six floating-point multipliers, and 13 floating-point comparators to achieve a throughput of one ray-AABB intersection test per cycle.

The IST unit performs ray-triangle intersection tests. Each IST unit consists of 11 floating-point adders, 11 floating-point multipliers, one reciprocal unit, and four floating-point comparators. The IST unit, like the TRV unit, includes an L1 cache and a ray accumulator buffer for effective memory access.

The ratio of TRV units to IST units is 16:1, which is different from that in the previous fixed ray-tracing pipelines [18], [19], [23], [25], [32] (3:1–4:1). The reason for this is that ray-primAABB tests in TRV units minimize the number of ray-triangle intersection tests.

Figure 6 illustrates the finite-state machine for processing of both BVH traversal and ray-primAABB intersection tests. Each state is described as follows.

• STAT_TRV (0) represents the initial traversal stage to fetch data. If the parent node is an inner node, we fetch the child node's AABB and the next state is set to STAT_LCHD (1) to visit the left child node. If the parent node is a leaf node, the next state is STAT_PRIM (4). If the traversal is finished, the next state is STAT_SHD (6).

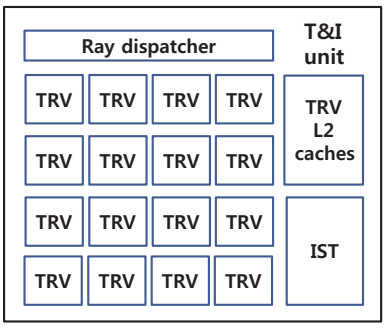


Fig. 5. The architecture of the traversal (TRV) and intersection (IST) unit. In contrast to the prior T&I engine [23], the ratio between TRV and IST units is 16:1 with the ray-primAABB intersection scheme, which reduces the expense of IST units per T&I unit.

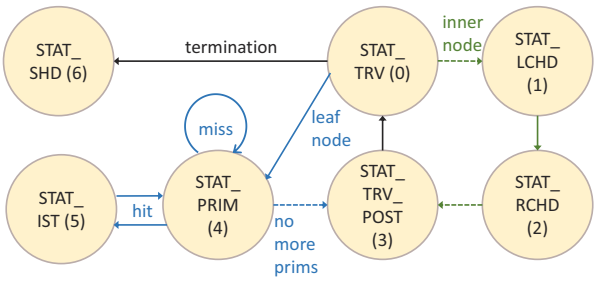


Fig. 6. The finite-state machine of the traversal unit. A dash arrow, such as 0→ 1, 2→ 3, and 4→ 3, means a state transition in the same iteration without additional shape data fetch.

• STAT_LCHD (1) performs the left child traversal. After the ray-AABB intersection test, the next state is set to STAT_RCHD (2).

• STAT_RCHD (2) performs the right child traversal. After the ray-AABB intersection test, the next state is set to STAT_TRV_POST (3).

• STAT_TRV_POST (3) determines the next visit node. In this state, stack operations based on the short-stack algorithm with the restart-trail method [47] are performed. Additionally, the SATO metric [48] is adapted to accelerate shadow ray traversal. First, if a non-shadow ray and a shadow ray intersect both of the child nodes, the next node is determined based on the child node of the nearest node and of the node with the lower cost based on SATO metric, respectively. The other node is pushed into the stack and the restart-trail flag bit at the current tree level is updated. If the stack is full, the bottom-most entry is discarded. After that, the current tree level is increased for further restart-trail updates. Second, if a ray intersects either the left child node or the right child node, only the current tree level is increased. Third, if the ray does not intersect either child node, the next visit node is popped from the stack and the current level bit is decreased. If the stack is empty, the node traversal is restarted from the root node. In this case, the restart-trail flag prevents duplicated visits to already traversed sub-trees. Because the next traversal step corresponds to a visit to the nodes in the above three cases, we set the next state to STAT_TRV (0).

• STAT_PRIM (4) performs ray-primAABB intersection tests. If the ray passes the test, the next state is STAT_IST (5). Additionally, if there are remaining primAABBs for further intersection in the leaf node, the processing is iterated with the current state (STAT_PRIM (4)). When we find the hit point of an occluded ray or have visited all primAABBs in the leaf, we change the state into STAT_TRV_POST (3).

• STAT_IST (5) passes the ray into the IST unit.

• STAT_SHD (6) passes the ray into the shaders when the final hit point of the ray is found.

3.5 Cache-Data Reuse Scheme

Efficient memory access is important for high performance when rays are incoherent. For incoherent ray tracing, we present a cache-data reuse scheme to exploit consecutive access. The block size of an L1 traversal cache is 64 bytes, so two sets of BVH node data or primAABB data can be stored in a cache block. In the case of BVH node data, left and right child nodes are stored consecutively. PrimAABBs in a leaf node are also stored consecutively. Therefore, we can reuse the cache-block data for the next iteration after the data are obtained. From the L1 cache, we obtain two sets of shape data (node or primAABB data) in an entire cache block and continuously maintain these data in the next pipeline stages. After dozens of cycles, when the ray comes back to the top of the traversal pipeline for the next iteration, we can reuse the shape data. If the required shape data exist in the maintained cache-block data, a cache access for the shape data is bypassed and the processing of the ray is treated as a cache hit.

4 Simulation Results and Analysis

In this section, we describe the experimental results using a cycle-accurate simulator. This simulator provides all of the cycles required for BV refitting and ray-tracing, hardware utilization, average T&I operations per ray, cache/memory statistics, and simulated performance, etc. We also compare our system to other approaches and describe the limitations of our work.

4.1 The Effect of the Ray-PrimAABB Test Scheme and Asynchronous BVH Construction

We used four dynamic test scenes for the experiment (Figure 7): UNC cloth simulation (92K triangles), Fairy forest (174K triangles), Exploding Dragon (252K triangles), and Lion (1.6M triangles). The Cloth scene has high frame-to-frame coherence, so it is suitable for BV refitting. The Fairy scene is used for game-like scene configuration. The Dragon scene has low frame-to-frame coherence due to fractures after a collision between a bunny and a dragon. The Lion scene is the largest scene in our benchmark and has features similar to the Dragon scene. All scenes were rendered at 1920×1200 resolution. For our experiments, we used two different ray settings: ray casting with hard shadows with one light source and two-bounce forced specular reflection. The BVHs were constructed by using the binned SAH method [5]. All experiments were performed using a 3.5GHz Core i7 4770K CPU with 8GB of RAM.

To construct shallow BVHs, we modified the ratio of the expected traversal cost (K_T) to the expected intersection cost (K_I) when we calculated the SAH cost in [5]. When we did not use the ray-primAABB

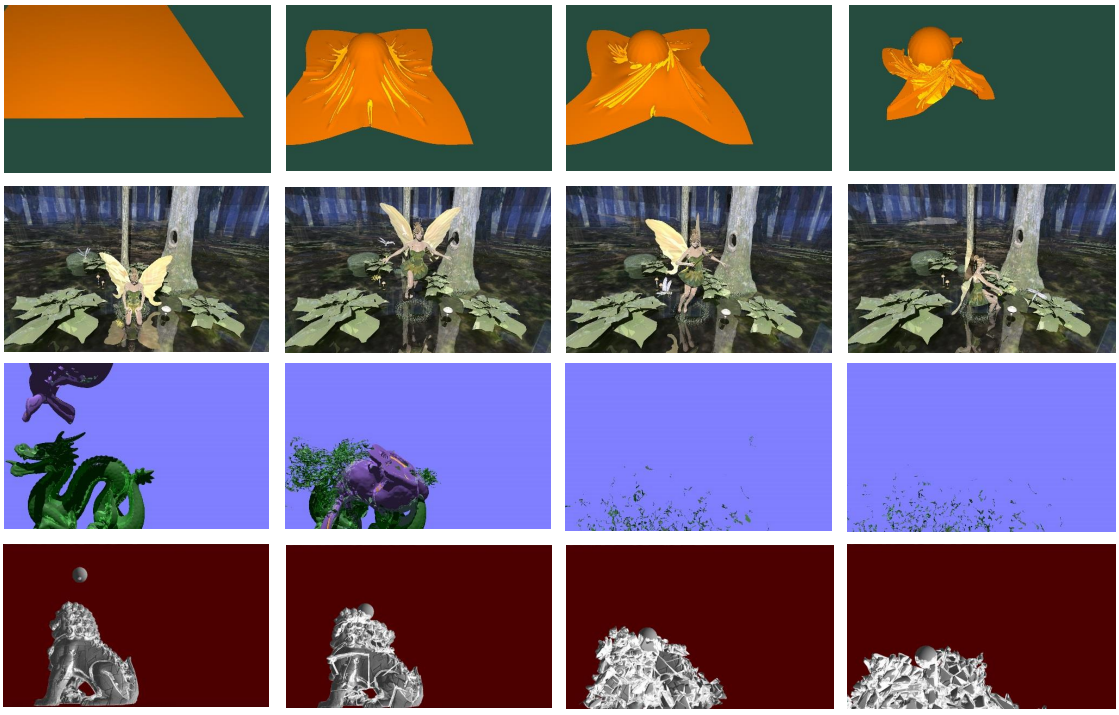


Fig. 7. Four dynamic test scenes (from top to bottom): Cloth, Fairy, Dragon, and Lion. All captured images were rendered with two-bounce forced specular reflection and hard shadows. According to our simulation results, our architecture can render these scenes at 264, 36, 212, and 46 FPS at 1920×1200 resolution, respectively.

test scheme, both the K_T and K_I values were set to 1. When we enabled the ray-primAABB test scheme, we varied the K_T values (2, 5, and 10) and set the K_I value to 1. The use of larger K_T values produces shallower trees having larger leaf nodes. The first goal of the experiments in this section is to determine the optimal ratio of K_T to K_I for our system.

Table 1 describes the results of our experiments. To measure the BVH build time, we use the multi-threaded BVH builder in Embree 2.3.2 [11]. We used four threads and one thread for the Lion scene and the other scenes, respectively. We believe this 1-4 thread setting is affordable in terms of resource utilization because modern CPUs, such as Intel Core i7 used in our experiments, can support up to eight threads.

The results in Table 1 are described as follows. First, the BVH build time decreases by 17–20%, 36–44%, and 46–50% when K_T is 2, 5, and 10, respectively. When we consider a small overhead for key-frame animation and data transfer, the BVH can be rebuilt at 25, 12, 9, and 3 times per second for the Cloth, Fairy, Dragon, and Lion scenes, respectively. Next, the number of intersection tests decreases 16–29% when the ray-primAABB intersection scheme has been enabled, but when K_T is 10, the number of traversal operations increases up to 27%. Thus, we conclude that the optimal K_T:K_I ratio with the ray-primAABB test scheme is 5:1 because this ratio provides the best performance balance between tree construction and ray traversal. Because the sum of T&I operations with the K_T values of 1 and 5 are similar, we think both settings will result in similar memory traffic.

TABLE 1
The experimental results for the ray-primAABB intersection scheme. For the statistics in this table, we selected a middle frame of each scene and rendered the scene with two-bounce reflection and shadows.

K_T:K_I	Avg tris per leaf	BVH build time (ms)	Avg TRV steps per ray	Avg IST steps per ray
Cloth (92K triangles) / single-threaded BVH build				
1:1	1.75	65 (1.00×)	33.46 (1.00×)	1.93 (1.00×)
2:1	2.84	52 (0.80×)	33.00 (0.99×)	1.37 (0.71×)
5:1	7.45	36 (0.56×)	34.87 (1.04×)	1.38 (0.72×)
10:1	14.49	33 (0.51×)	40.04 (1.20×)	1.38 (0.72×)
Fairy (174K triangles) / single-threaded BVH build				
1:1	2.06	120 (1.00×)	64.99 (1.00×)	6.82 (1.00×)
2:1	3.17	100 (0.83×)	63.73 (0.98×)	5.68 (0.83×)
5:1	7.35	77 (0.64×)	66.40 (1.02×)	5.72 (0.84×)
10:1	14.50	65 (0.54×)	78.90 (1.21×)	5.74 (0.84×)
Dragon (252K triangles) / single-threaded BVH build				
1:1	1.72	178 (1.00×)	51.63 (1.00×)	1.36 (1.00×)
2:1	2.80	142 (0.80×)	51.31 (0.99×)	0.98 (0.72×)
5:1	6.47	105 (0.59×)	52.63 (1.02×)	0.99 (0.73×)
10:1	12.24	88 (0.50×)	57.04 (1.10×)	1.00 (0.74×)
Lion (1.6M triangles) / parallel BVH build (4 threads)				
1:1	1.80	474 (1.00×)	85.39 (1.00×)	11.68 (1.00×)
2:1	2.82	390 (0.82×)	84.97 (1.00×)	8.86 (0.76×)
5:1	7.30	287 (0.61×)	91.27 (1.07×)	8.90 (0.76×)
10:1	14.40	246 (0.52×)	108.17 (1.27×)	8.94 (0.77×)

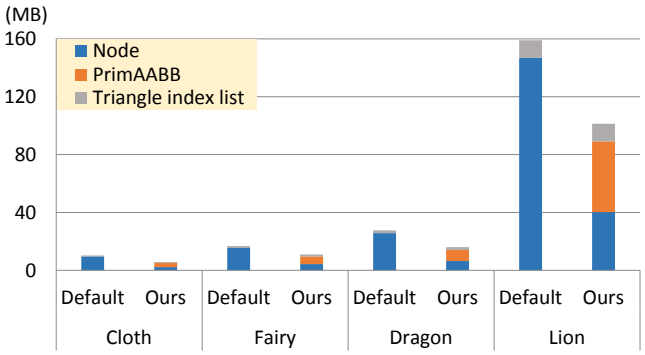


Fig. 8. Comparison of the tree sizes (in megabytes). Default – the ratio of K_T to K_I was 1:1 without the ray-primAABB intersection scheme. Ours – the ratio of K_T to K_I was 5:1 with the ray-primAABB intersection scheme.

Figure 8 depicts the tree sizes. Node data require triple buffering, and the size of a node is 32 bytes. Both primAABB data and a triangle index list require double buffering; PrimAABB data and a triangle index list do not need a buffer to store transferred data from the CPUs and a buffer in T&I units, respectively. Note that we do not count 8 bytes of primAABB data stored in the padding of TriAccel data (Figure 3) for these tree sizes in Figure 8.

According to Figure 8, the proposed method achieves a reduction of 34–44% of the tree sizes; the number of nodes is 72–77% less than the default setting, and primAABB data are added. The result also means that the bus-bandwidth requirements to transfer node and triangle index data from the CPU to the ray-tracing hardware are reduced by 66–71%. If the data are asynchronously transferred from the CPU to the ray tracing hardware, the required bus-bandwidth for BVHs is very small (22–52 MB/s) because these data do not need to be transferred during each frame.

4.2 Hardware Complexity and Area Estimation

The hardware setup of the proposed architecture is structured as follows. The number of stacks per TRV is 32, and the number of TRVs in a T&I unit is 16; therefore, the highest number of executing rays in a T&I unit is 512. We configured the external memory for 1GHz, 8-channel GDDR3 memory. We assumed that six channels are connected to T&I units, and two channels are connected to the GTU units. The memory simulation was executed using a GDDR3 simulator in GPGPU-Sim [49]. As with [23], we assumed that the programmable shaders provide sufficient computing power for ray generation and shading.

Table 2 shows the hardware complexity of a GTU unit and a T&I unit. Each BV refit, TRV, and IST unit has a 8KB, 16KB, and 128KB 2-way set associative cache, respectively. All caches have one read-only port. An L2 TRV cache is a 512KB 4-way cache divided into eight banks. The BV refit cache has a block size of 256B for data prefetching with a sequential access pattern. In contrast, both TRV and IST caches have a block size of 64B.

We set the latency of the L1 caches as one cycle and set the minimum latency of the L2 caches as 3 cycles. This configuration is the same as that in 500MHz MIMD TM [29] based on CACTI [50], and also corresponds to the cache latencies on AMD Opteron X4 (3-cycle L1 and 9-cycle L2 latencies at 2.5GHz) [51]. Additionally, caches in our architecture are read-only

TABLE 2
Complexity of each component by the number of floating-point units and the required on-chip memory. Abbreviations: ADD – adder, MUL – multiplier, RCP – reciprocal unit, CMP – comparator, L1 – L1 cache, L2 – L2 cache, idx. – index, vtx. – vertex, calc. – calculation.

	ADD	MUL	RCP	CMP	RF	L1	L2
GTU unit							
Idx. fetch					2KB		
Vtx. fetch					5KB		
Vtx. interp.	10	5			1KB		
TriAccel/ AABB calc.	10	12	1	7	20KB		
BV refit				11	2KB	8KB	
Total	20	17	1	18	30KB	8KB	
T&I unit							
1 RD			3		2KB		
16 TRV	96	96		208	207KB	256KB	512KB
1 IST	11	11	1	3	13KB	128KB	
I/O buffer					32KB		
Total	107	107	4	211	254KB	384KB	512KB

TABLE 3
Area estimates of a GTU unit and a T&I unit. Abbreviations: FP – floating-point, INT – integer.

Functional Unit	Area (mm^2)	Total Area (mm^2)	Memory Unit	Area (mm^2)	Total Area (mm^2)
GTU unit					
FP ADD	0.003	0.06	BV refit		0.03
FP MUL	0.01	0.17	4K RFs	0.019	0.14
FP RCP	0.11	0.11			
FP CMP	0.00072	0.01			
INT ADD	0.00066	0.02			
Control/Etc.		0.09			
Wiring overhead					0.54
Total					1.07
T&I unit					
FP ADD	0.003	0.32	TRV L1	0.037	0.60
FP MUL	0.01	1.07	TRV L2		1.23
FP RCP	0.11	0.44	IST		0.25
FP CMP	0.00072	0.08	4K RFs	0.019	1.18
INT ADD	0.00066	0.03			
Control/Etc.		0.45			
Wiring overhead					3.88
Total					9.51

in contrast to modern CPUs/GPUs, so we did not need to consider complex cache coherency issues. We also considered bank conflicts; if a bank conflict in an L2 cache occurs, the L2 cache access is delayed. As a result, the actual L2 cache latency is usually more than 10 cycles with coherent rays; if rays are very incoherent, the L2 cache latency can increase by up to dozens of cycles.

The latencies of a floating-point (FP) multiplier, an FP adder, an FP comparator, and a reciprocal unit were set to 2, 2, 1, and 16 cycles, respectively, similar to [29]. Register files (RFs) are needed for buffers between the units, ray accumulation buffers, 8-entry traversal stacks, I/O buffers to programmable shaders, and pipeline registers. The size of a ray accumulation buffer is 32 (2 (width) $\times$ 16 (height)).

To predict the performance of our system, we carefully estimated the area of a GTU unit and a T&I unit (Table 3), using an estimation metric similar to that in [23]. First, we assumed 65 nm technology, a 200 mm^2 die area, and a clock speed of 500 MHz, similar to TRaX [28]. Second, we assumed that the GTU unit and T&I units occupy less than 33% of the total area, similar to D-RPU [19]. The remaining area was used for programmable shaders and memory interfaces. Third, we used the area estimates for arithmetic units and caches obtained from [29] and CACTI 6.5 [50]. Fourth, we assumed that control parts require 23% of the total area for arithmetic units; this assumption is based on the ratio of the front-to-end area to that of the execution area in [52]. Fifth, we added 69% overhead into our estimation. This was assumed by two levels of wiring overhead (arithmetic units $\rightarrow$ each component $\rightarrow$ a GTU unit and a T&I unit); the one-level overhead used in [52] is approximately 30%. According to these estimates, four GTU units (4.3 mm^2) and six T&I units (57.1 mm^2) can be assigned into a ray-tracing core with a 200 mm^2 die area (31% of the total area).

4.3 Simulation Results in Dynamic Scenes

For experiments with dynamic scenes, we used the same test scenes and experimental setup as in Section 4.1. Table 4 describes the results: our system can achieve real-time frame rates at 1920$\times$1200 resolution. The performance effects of two-bounce reflection are different in each scene due to the different required cycles for BV update and ray tracing changed in each scene (Figure 9). In the Fairy scene, frame rates with a ray recursion depth of 2 are 3$\times$ lower than that with a depth of 0 because almost all radiance rays (primary and reflection rays) hit some objects and generate additional rays. In contrast, many radiance rays in the other scenes do not hit any objects (background colors in Figure 7) and do not propagate additional rays. Thus, the differences of frame rates between the ray recursion depths of 0 and 2 in these scenes are less

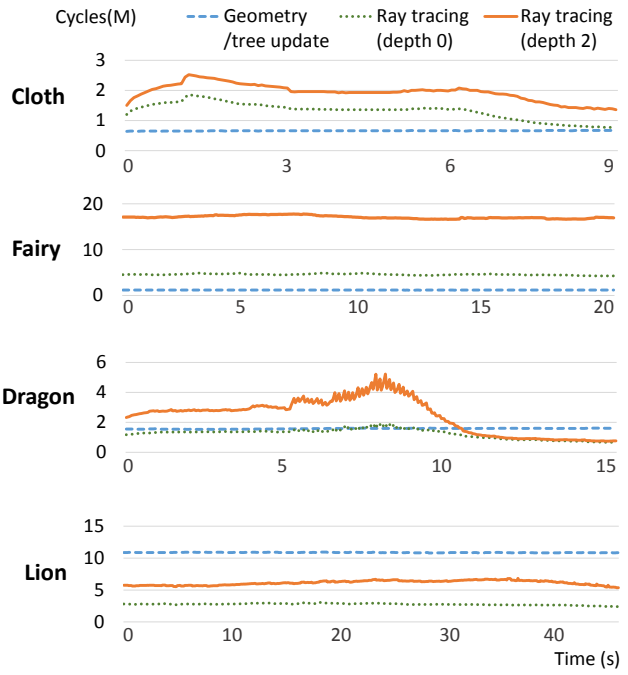


Fig. 9. Required cycles per frame for the test scenes.

than those in the Fairy scene. In particular, the Lion scene shows almost same frame rates regardless of the ray recursion depths because the geometry and tree update time is bottlenecked in this scene due to its high triangle count (1.6M). However, real-time frame rates (46 FPS) are still shown.

Memory traffic per frame is broadly proportional to the scene size because more triangles require more memory accesses for geometry and tree updates. Additionally, two-bounce reflection increases memory traffic due to low cache hit rates; various directions of normal vectors of each object can result in incoherent reflection rays which can decrease cache hit rates.

In Table 5, we compare our system to other approaches. For a comparison with a CPU approach, we executed the Manta ray tracer [21] with the tree-rotation algorithm [9]. For ray traversal, we used the DynBVH traversal algorithm [4] with an 8$\times$8 packet size. The result on a 3.5GHz Core i7 CPU is 8–23 FPS. For comparison with a GPU approach, we used NVIDIA OptiX 3.6.2 [22]. For key-frame animation, we modified the Sample6 code in OptiX SDK. We measured performance with an NVIDIA GeForce GTX680 card, which gave a result of 7–30 FPS. In contrast to these CPU and GPU approaches, our architecture can achieve real-time frame rates in all the test scenes. These high frame rates are due to high ray traversal performance, the maintenance of tree quality, and a low tree-update overhead.

We have also implemented a CPU-GPU hybrid ray tracer based on asynchronous BVH construction. The detailed description of the hybrid ray tracer is beyond the scope of this paper and is included in another

TABLE 4
Simulation results in dynamic scenes.

Scene (max ray depth)	# of rays per frame (M)	Cache hit rate (%) (BV Refit/TRV L1/ TRV L2/IST)	Average TRV/IST steps per ray	Memory traffic (MB/frame)	Simulated frames per second
Cloth (0)	3.0	86 / 97 / 95 / 98	28.8 / 1.3	37.0	402
Cloth (2)	3.9	86 / 95 / 95 / 97	33.6 / 1.4	51.3	264
Fairy (0)	4.6	85 / 98 / 96 / 99	63.1 / 4.3	63.0	109
Fairy (2)	11.9	85 / 96 / 94 / 98	66.5 / 5.6	187.7	36
Dragon (0)	2.6	85 / 98 / 93 / 95	28.9 / 0.5	80.0	312
Dragon (2)	3.3	85 / 93 / 84 / 85	41.0 / 0.8	166.5	212
Lion (0)	2.7	85 / 92 / 97 / 98	54.2 / 4.7	763.3	46
Lion (2)	3.6	85 / 88 / 94 / 96	86.4 / 8.1	923.2	46

TABLE 5
Comparison of the performance for ray casting with shadows at 1920×1200 resolution.

	CPU (Manta) [21]	GPU (OptiX) [22]	MIC [17]	CPU-GPU hybrid [53]	Ours
Platform	Intel Core i7 4770K (4 cores)	NVIDIA GeForce GTX 680 (1536 cores)	Intel MIC (32 cores)	Intel i7 4770K + NVIDIA GTX 680	RT H/W + CPU (1-4 cores)
Clock (MHz)	3500	1006	1000	3500(CPU) & 1006(GPU)	500 (RT H/W only)
Process (nm)	22	28	45	22(CPU) & 28(GPU)	65 (")
Area (mm^2)	177	294	-	177(CPU) & 294(GPU)	200 (")
BVH update method	BV refitting [4] + tree rotation [9]	LBVH [13] + BVH refinement [16]	Binned SAH BVH construction [17]	Asynchronous BVH construction [1]	
FPS (Cloth)	23	30	44	35	402
FPS (Fairy)	8	17	17	25	109
FPS (Dragon)	18	26	19	39	312
FPS (Lion)	8	7	-	18	46

technical report [53]. The results on a 3.5GHz Intel Core i7 CPU and an NVIDIA GTX680 GPU are 18-35 FPS, and our architecture is at least 2.5× faster than this CPU-GPU hybrid ray tracer.

Compared to full SAH BVH construction on the Intel MIC architecture [17], our approach takes advantage of asynchronous BVH construction and heterogeneous computing environments. According to [17], full BVH construction spent 41–65% of the total rendering time in the Cloth, Fairy, and Dragon scenes. In contrast, the GTU unit in our architecture occupies less than 3% of the total die area and an existing CPU performs tree reconstruction.

Fig. 10. Sample images from the three static test scenes: Sibenik rendered with ray casting, Fairy rendered with ambient occlusion, and Conference rendered with diffuse inter-reflection.

4.4 Simulation Results in Static Scenes

Our ray-tracing system can also be used to accelerate the rendering of static scenes. To measure the performance of our architecture in static scenes, we set up the following experimental environment, similar to [23], [46]. We used the three scenes in Figure 10: Sibenik (80K triangles), Fairy Forest (174K triangles), and Conference (282K triangles). We obtained ray data from Aila's CUDA ray tracer [46]. The resolution is 1024×768 and the ray types are the primary ray (very coherent), the ambient occlusion (AO) ray (incoherent), and the diffuse inter-reflection ray (very incoherent). The number of samples per pixel is 32. We used AO cut-off values of 5.0, 0.3, and 5.0 for the Sibenik, Fairy, and Conference scenes, respectively. We used the same view points as [46] and the performance values are averages from five representative viewpoints per scene. The BVHs were built by the split BVH build algorithm [54]. Note that we assumed that eight memory channels are connected to T&I units in contrast to Section 4.3. The reason is that we investigate not dynamic scene performance but ray traversal performance, in this section.

Table 6 summarizes the results in the static test scenes. According the results, our ray-tracing system achieves 351–969 Mrays/s. Compared to a kd-tree-based ray-tracing hardware architecture for static scenes [23], our proposed architecture performs at an average 94.4% of [23] and these results are comparable to those of the kd-tree-based architecture. Additionally, our architecture performs better than GPU ray tracing on GTX680 [55], even though our architecture requires less computational resources than do modern

TABLE 6
Simulation results in static scenes.

Ray type	TRV/IST utilization (%)	Cache hit rate (%) (TRV L1/TRV L2/IST)	Average TRV/IST steps per ray	Memory traffic (GB/s)	Simulated Mrays/s	Relative performance compared to [23]
Sibenik (80K triangles)						
Primary	95 / 56	99 / 93 / 99	65.4 / 2.9	1.8	588	128%
AO	95 / 53	97 / 99 / 99	43.7 / 1.8	0.4	928	113%
Diffuse	65 / 50	87 / 92 / 89	76.9 / 4.3	27.6	351	80%
Fairy (174K triangles)						
Primary	73 / 72	98 / 93 / 99	89.7 / 8.7	2.7	383	106%
AO	64 / 80	96 / 97 / 99	45.0 / 4.3	2.6	649	80%
Diffuse	59 / 81	89 / 91 / 94	69.8 / 7.2	21.9	380	102%
Conference (282K triangles)						
Primary	82 / 73	99 / 93 / 99	58.9 / 3.7	1.9	602	76%
AO	90 / 51	97 / 98 / 99	38.8 / 1.6	1.2	969	82%
Diffuse	75 / 77	91 / 91 / 92	61.3 / 4.6	24.1	506	84%

TABLE 7
The effect of the cache-data reuse scheme for L1 traversal caches.

Scene	TRV/IST utilization	TRV L1/ L2 cache hit (%)	Simulated Mrays/s
Without our cache scheme			
Sibenik	53 / 41	84 / 92	290
Fairy	55 / 75	87 / 91	349
Conference	68 / 70	89 / 92	462
With our cache scheme			
Sibenik	65 / 50	87 / 89	351
Fairy	59 / 81	89 / 91	380
Conference	75 / 77	91 / 91	506

desktop GPUs, as described in Table 5. In particular, when tracing incoherent rays, our MIMD architecture results in less performance degradation than modern SIMT-based GPU architectures.

We also investigated the efficiency of the cache scheme presented in Section 3.5. For this experiment, we traced diffuse inter-reflection rays, which are the most incoherent ray type in our benchmark. Under the cache-data reuse scheme, a ray that bypassed an L1 traversal cache was counted as a cache hit for the cache hit-rate calculation. According to the results shown in Table 7, the cache-data reuse scheme for L1 cache access improves the ray-tracing performance by 9–21% with increased cache hit rates. In terms of chip area, the cache-data reuse scheme requires additional register spaces to store a 64B cache block of data in each pipeline stage and buffer. However, additional registers for the T&I unit, which are 28KB, require only $0.13mm^2$.

4.5 Discussion and Limitations

More effective BVH update: Our system's performance may drop off in complex dynamic scenes because of a long tree-build time and large memory footprints. In addition, our system would not be suitable for very rapidly-changing scenes (e.g., racing) because asynchronous BVH construction exploits frame-to-frame coherence. Additionally, object insertion, object deletion, or completely unstructured motion with topological changes can generate a frame drop in asynchronous BVH construction [1], since in these situations the entire BVH should be reconstructed in every frame. Finally, once triple buffering is used, the tree data stored in the first and second buffers are two and one frame old, respectively. The possibility of performance degradation caused by the outdated data in the first buffer was described above, but an early finish of BV refitting using the second buffer can result in delayed rendering. For example, if a significant amount of time is still required for ray tracing during the current frame after BV refitting has finished, the BVH in the second buffer will be outdated during the idle time of GTU units. The outdated BVH can make a perceptible delay if frame rates are very low.

We think there are four possible future improvements: partial update, tree rotation, faster BVH construction, and continuous BV refitting operations. First, if we divide the static parts and dynamic parts of the tree using multi-level hierarchies like gkDtrees [8], we can more effectively render dynamic scenes, which mainly consist of static parts. Since we do not need to rebuild and refit static parts, and these static parts only need a single buffer, this method will alleviate the problem of the rapid BVH rebuild time outdating the rebuilt BVH. In case of object insertion, object deletion, or topological changes, the dynamic parts can be selectively restructured, in a similar manner to the method in [6]. Second, if the BV refit unit is extended to support the tree-rotation algorithm [9], the BVH update will be more robust for rapidly-changing scenes. Because the tree-rotation algorithm can be easily integrated to BVH refitting, we believe this addition can be available in our hardware architecture. Third, our approach can be combined with faster ways of constructing the BVH, such as the

approximate agglomerative clustering algorithm on multi-core CPUs [10] or a dedicated hardware unit for BVH construction [20]. In this case, since the tree need not be reconstructed during each frame, our small GTU unit will help to reduce the required hardware resources for tree construction. Finally, continuous BV refitting operations can alleviate the delayed rendering problem. After BV refitting computation is finished, we can update the BVH again at the current time if a long time remains to finish ray tracing at the current frame. This approach will generate a recently updated BVH.

Shading cost: Shading may or may not be a major cost in ray tracing [46], but we assumed that shading would not be a bottleneck due to the shading cost of Whitted ray tracing and the sufficient capability of programmable shaders. In the Embree system, complex shading for off-line rendering typically consumes 30 to 50% of the total frame time [11]; it indicates that simpler shading for real-time ray tracing dynamic scenes would result in cost lower than 30% of the total frame time. In fact, shading costs in simpler scenes occupy about 25% of the total rendering time in Wald's experiments (Table 7.6 in [44]). Additionally, a GPU ray tracer [55] on GTX680 shows very high ray generation and shading (RGS) performance (several hundred million rays per second). Even a cycle-accurate simulation result on a state-of-the-art mobile processor (4-core SRP) exhibited RGS performance by up to 198 M rays/s [32]. However, complex shading would bottleneck rendering. In this case, we think additional stream filter units [27] would be a suitable solution to maintain high SIMD utilization of the programmable shaders.

Animation and primitive types: Because we focused on a very small fixed unit for dynamic scenes, our architecture currently supports triangular primitives and key-frame animations. To support other animation and primitive types, appropriate programmable shaders would be needed. In regards to animation types, the usage of GTU units will be different. If hierarchical transformation or skinned animation is performed on shaders, the interpolation unit for key-frame animation in a GTU unit will not be used. If an object is inserted or deleted in a scene, or the geometry/tessellation shader increases the number of triangles in an object, the tree data of the object in the GTU unit are invalid and so should be reconstructed on a CPU. In this case, selective restructuring [6] or multi-level tree decomposition [8], [11], [56] can be used to effectively handle the dynamic objects as described above. In regards to primitive types, more complex interfaces between TRV units and programmable shaders will be required to prevent performance degradation caused by frequent communication or unbalanced workloads between traversal and intersection operations.

5 Conclusions and Future Work

In this paper, we have presented a hybrid ray-tracing architecture for dynamic scenes. Our approach achieves real-time frame rates using asynchronous BVH construction [1], [7] on a CPU, and dedicated ray-tracing hardware. We have also presented a novel traversal hardware architecture using ray-primAABB tests and an efficient cache scheme for the architecture.

There are many avenues for future work. First, new acceleration data structures would help to increase the performance of our architecture. Our BVH-based hardware architecture is slower than a kd-tree-based hardware architecture [23] in some static scenes, as described in Section 4.4; we think that an extended architecture based on shared-plane BVHs (SPBVHs) [57] can compensate for the defect because SPBVHs have a lower traversal cost and memory footprints than BVHs. Second, we would like to prove the feasibility of our system on a register-transfer-level (RTL) implementation, after which we would like to run experiments on actual hardware. Integration with mobile GPU architectures [32] will be especially helpful for new killer mobile applications. Finally, we are interested in extending our architecture to accelerate ray-tracing-based sound rendering [58].

Acknowledgments

This work was supported by Samsung Electronics Co., Ltd. Jae-Ho was also supported by the National Research Foundation of Korea Grant funded by the Korean Government (Ministry of Education) [NRF-2012R1A6A3A03040332]. Dinesh Manocha was supported by ARO Contract W911NF-10-1-0506, and NSF awards 0917040 and 1320644. Models used are courtesy of the UNC Dynamic Scene Benchmarks (Cloth Simulation, Exploding Dragon, and Lion), the Utah 3D Animation Repository (Fairy Forest), Marko Dabrovic (Sibenik), and Anat Grynberg and Greg Ward (Conference). We would like to thank Tero Karras, Timo Aila, and Samuli Laine for releasing their GPU ray tracer, and to thank Tor Aamodt and his lab members for releasing GPGPU-Sim.

References

[1] T. Ize, I. Wald, and S. G. Parker, "Asynchronous BVH construction for ray tracing dynamic scenes on parallel multi-core architectures," in *In Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, 2007, pp. 101–108.

[2] I. Wald, W. R. Mark, J. Gunther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley, "State of the art in ray tracing animated scenes," *Computer Graphics Forum*, vol. 28, no. 6, pp. 1691–1722, 2009.

[3] C. Lauterbach, S.-E. Yoon, D. Tuft, and D. Manocha, "RT-DEFORM: Interactive ray tracing of dynamic scenes using BVH," in *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, 2006, pp. 39–45.

[4] I. Wald, S. Boulos, and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchies," *ACM Transactions on Graphics*, vol. 26, no. 1, pp. 6:1–6:18, 2007.

[5] I. Wald, "On fast construction of SAH-based bounding volume hierarchies," in *Proceedings of IEEE Symposium on Interactive Ray Tracing 2007*, 2007, pp. 33–40.

[6] S.-E. Yoon, S. Curtis, and D. Manocha, "Ray tracing dynamic scenes using selective restructuring," in *Proceedings of Eurographics symposium on rendering 2007*, 2007, pp. 73–84.

[7] I. Wald, T. Ize, and S. G. Parker, "Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes," *Computers & Graphics*, vol. 32, no. 1, pp. 3–13, 2008.

[8] Y.-S. Kang, J.-H. Nah, W.-C. Park, and S.-B. Yang, "gkDtree: A group-based parallel update kd-tree for interactive ray tracing," *Journal of Systems Architecture*, vol. 59, no. 3, pp. 166–175, 2013.

[9] D. Kopta, T. Ize, J. Spjut, E. Brunvand, A. Davis, and A. Kensler, "Fast, effective BVH updates for animated scenes," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2012, pp. 197–204.

[10] Y. Gu, Y. He, K. Fatahalian, and G. Blelloch, "Efficient BVH construction via approximate agglomerative clustering," in *Proceedings of the 5th High-Performance Graphics Conference*, 2013, pp. 81–88.

[11] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst, "Embree - a kernel framework for efficient CPU ray tracing," *ACM Transactions on Graphics (SIGGRAPH 2014)*, vol. 33, no. 4, pp. 143:1–143:8, 2014.

[12] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time KD-tree construction on graphics hardware," *ACM Transactions on Graphics (SIGGRAPH Asia 2008)*, vol. 27, no. 5, pp. 1–11, 2008.

[13] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH construction on GPUs." *Computer Graphics Forum (EUROGRAPHICS 2008)*, vol. 28, no. 2, pp. 375–384, 2009.

[14] K. Garanzha, J. Pantaleoni, and D. McAllister, "Simpler and faster HLBVH with work queues," in *Proceedings of the Conference on High Performance Graphics*, 2011, pp. 59–64.

[15] T. Karras, "Maximizing parallelism in the construction of BVHs, octrees, and k-d trees," in *Proceedings of the 4th conference on High-Performance Graphics*, 2012, pp. 33–37.

[16] T. Karras and T. Aila, "Fast parallel construction of high-quality bounding volume hierarchies," in *Proceedings of the 5th High-Performance Graphics Conference*, 2013, pp. 89–99.

[17] I. Wald, "Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 1, pp. 47–57, 2012.

[18] S. Woop, G. Marmitt, and P. Slusallek, "B-KD trees for hardware accelerated ray tracing of dynamic scenes," in *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 2006, pp. 67–77.

[19] S. Woop, E. Brunvand, and P. Slusallek, "Estimating performance of a ray-tracing ASIC design," in *Proceedings of the 2006 IEEE/EG Symposium on Interactive Ray Tracing*, 2006, pp. 7–14.

[20] M. J. Doyle, C. Fowler, and M. Manzke, "A hardware unit for fast SAH-optimised BVH construction," *ACM Transactions on Graphics (SIGGRAPH 2013)*, vol. 32, no. 4, pp. 66:1–66:13, 2013.

[21] J. Bigler, A. Stephens, and S. G. Parker, "Design for parallel interactive ray tracing systems," in *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, 2006, pp. 187–196.

[22] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "OptiX: a general purpose ray tracing engine," *ACM Transactions on Graphics (SIGGRAPH 2010)*, vol. 29, no. 4, pp. 66:1–66:13, 2010.

[23] J.-H. Nah, J.-S. Park, C. Park, J.-W. Kim, Y.-H. Jung, W.-C. Park, and T.-D. Han, "T&I engine: traversal and intersection engine for hardware accelerated ray tracing," *ACM Transactions on Graphics (SIGGRAPH Asia 2011)*, vol. 30, no. 6, pp. 160:1–160:10, 2011.

[24] C. Lauterbach, Q. Mo, and D. Manocha, "gProximity: Hierarchical GPU-based operations for collision and distance queries," *Computer Graphics Forum (EUROGRAPHICS 2010)*, vol. 29, no. 2, pp. 419–428, 2010.

[25] J. Schmittler, S. Woop, D. Wagner, W. J. Paul, and P. Slusallek, "Realtime ray tracing of dynamic scenes on an FPGA chip," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2004, pp. 95–106.

[26] S. Woop, J. Schmittler, and P. Slusallek, "RPU: a programmable ray processing unit for realtime ray tracing," *ACM Transactions on Graphics (SIGGRAPH 2005)*, vol. 24, no. 3, pp. 434–444, 2005.

[27] K. Ramani, C. P. Gribble, and A. Davis, "StreamRay: a stream filtering architecture for coherent ray tracing," in *ASPLOS '09: Proceeding of the Architectural support for programming languages and operating systems*, 2009, pp. 325–336.

[28] J. Spjut, A. Kensler, D. Kopta, and E. Brunvand, "TRaX: a multicore hardware architecture for real-time ray tracing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 12, pp. 1802–1815, 2009.

[29] D. Kopta, J. Spjut, E. Brunvand, and A. Davis, "Efficient MIMD architectures for high-performance ray tracing," in *ICCD 2010: Proceedings of the 28th IEEE International Conference on Computer Design*, 2010, pp. 9–16.

[30] T. Aila and T. Karras, "Architecture considerations for tracing incoherent rays," in *HPG' 10: Proceedings of the Conference on High Performance Graphics*, 2010, pp. 113–122.

[31] W.-J. Lee, S.-H. Lee, J.-H. Nah, J.-W. Kim, Y. Shin, J. Lee, and S.-Y. Jung, "SGRT: a scalable mobile GPU architecture based on ray tracing," in *ACM SIGGRAPH 2012 Talks*, 2012.

[32] W.-J. Lee, Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, S.-Y. Jung, S.-H. Lee, H.-S. Park, and T.-D. Han, "SGRT: A mobile GPU architecture for real-time ray tracing," in *Proceedings of the 5th High-Performance Graphics Conference*, 2013, pp. 109–119.

[33] J.-H. Nah, H.-J. Kwon, D.-S. Kim, C.-H. Jeong, J. Park, T.-D. Han, D. Manocha, and W.-C. Park, "RayCore: A ray-tracing hardware architecture for mobile devices," *ACM Transactions on Graphics*, vol. 33, no. 5, pp. 162:1–162:15, 2014.

[34] S. Woop, "A programmable hardware architecture for real-time ray tracing of coherent dynamic scenes," Ph.D. dissertation, Sarrland University, 2007.

[35] B. C. Budge, J. C. Anderson, C. Garth, and K. I. Joy, "A hybrid CPU-GPU implementation for interactive ray-tracing of dynamic scenes," University of California, Davis Computer Science, Tech. Rep. CSE-2008-9, 2008.

[36] J.-H. Nah, Y.-S. Kang, K.-J. Lee, S.-J. Lee, T.-D. Han, and S.-B. Yang, "MobiRT: an implementation of OpenGL ES-based CPU-GPU hybrid ray tracer for mobile devices," in *ACM SIGGRAPH ASIA 2010 Sketches*, 2010, pp. 50:1–50:2.

[37] J. Bikker and J. van Schijndel, "The brigade renderer: A path tracer for real-time games," *International Journal of Computer Games Technology*, 2013.

[38] B. Budge, T. Bernardin, J. A. Stuart, S. Sengupta, K. I. Joy, and J. D. Owens, "Out-of-core data management for path tracing on hybrid resources," *Computer Graphics Forum*, vol. 28, no. 2, pp. 385–396, 2009.

[39] A. Pajot, L. Barthe, M. Paulin, and P. Poulin, "Combinatorial bidirectional path-tracing for efficient hybrid CPU/GPU rendering," *Computer Graphics Forum*, vol. 30, no. 2, pp. 315–324, 2011.

[40] LuxRender, "Luxrays." [Online]. Available: http://www.luxrender.net/wiki/LuxRays

[41] A. Reshetov, "Faster ray packets - triangle intersection through vertex culling," in *Proceedings of IEEE Symposium on Interactive Ray Tracing 2007*, 2007, pp. 105–112.

[42] J. Snyder and A. Barr, "Ray tracing complex models containing surface tessellations," in *ACM SIGGRAPH Computer Graphics*, vol. 21, 1987, pp. 119–128.

[43] J.-H. Nah, W.-C. Park, Y.-S. Kang, and T.-D. Han, "Ray-box culling for tree structures," *Journal of Information Science and Engineering*, vol. 29, no. 6, pp. 1211–1225, 2013.

[44] I. Wald, "Realtime ray tracing and interactive global illumination," Ph.D. dissertation, Sarrland University, 2004.

[45] T. Möller and B. Trumbore, "Fast, minimum storage ray-triangle intersection," *Journal of Graphics Tools*, vol. 2, no. 1, pp. 21–28, 1997.

[46] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on GPUs," in *HPG '09: Proceedings of the Conference on High Performance Graphics*, 2009, pp. 145–149.

[47] S. Laine, "Restart trail for stackless BVH traversal," in *HPG '10: Proceedings of the Conference on High Performance Graphics*, 2010, pp. 107–111.

[48] J.-H. Nah and D. Manocha, "SATO: Surface-area traversal order for shadow ray tracing," *Computer Graphics Forum*, vol. 33, no. 6, pp. 167–177, 2014.

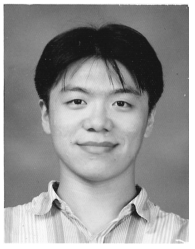
[49] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software 2009*, 2009, pp. 163–174.
[50] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 3–14.
[51] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 4th ed. Morgan Kaufmann Publishers Inc., 2008.
[52] A. Mahesri, D. Johnson, N. Crago, and S. J. Patel, "Tradeoffs in designing accelerator architectures for visual computing," in *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 164–175.
[53] J.-W. Kim, J.-M. Kim, M. Lee, and T.-D. Han, "Asynchronous BVH reconstruction on CPU-GPU hybrid architecture," in *ACM SIGGRAPH 2014 Posters*, 2014, pp. 91:1–91:1.
[54] M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies," in *HPG' 09: Proceedings of the Conference on High Performance Graphics*, 2009, pp. 7–13.
[55] T. Aila, S. Laine, and T. Karras, "Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum," NVIDIA Corporation, NVIDIA Technical Report NVR-2012-02, 2012.
[56] I. Wald, C. Benthin, and P. Slusallek, "Distributed interactive ray tracing of dynamic scenes," in *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 2003, pp. 77–86.
[57] M. Ernst and S. Woop, "Ray tracing with shared-plane bounding volume hierarchies," *Journal of Graphics, GPU, and Game Tools*, vol. 15, no. 3, pp. 141–151, 2011.
[58] C. Schissler, R. Mehra, and D. Manocha, "High-order diffraction and diffuse reflections for interactive sound propagation in large environments," *ACM Transactions on Graphics (SIGGRAPH 2014)*, vol. 33, no. 4, pp. 39:1–39:12, 2014.

Jae-Ho Nah received the B.S., M.S., and Ph.D. degrees from the Department of Computer Science, Yonsei University in 2005, 2007, and 2012, respectively. Currently, he is a senior research engineer at LG Electronics. His research interests include ray tracing, rendering algorithms, and graphics hardware.

Jin-Woo Kim is a PhD candidate of the Media System Laboratory at Yonsei University, Seoul, Korea. He received the BS degree from Sangmyung University, Seoul, Korea, in 2006. His research interests include 2D/3D graphics hardware, real-time ray tracing, and GPGPU based parallel programming.

Junho Park currently works at Humax, Korea. He received his BS and MS degrees from Soongsil University and Yonsei University, respectively. His main interests include ray tracing, GPGPU, and computer vision.

Won-Jong Lee is a senior researcher of the Processor Architecture Lab at Samsung Advanced Institute of Technology. He received his PhD and M.S. degree in computer science from Yonsei University, Seoul, Korea, in 2001 and his B.S. degree in computer engineering from Inha University, Incheon, Korea, in 1999. His research interests include mobile GPU, graphics hardware, ray tracing, parallel and distributed rendering. Currently he is leading a project on designing a mobile GPU architecture based on ray tracing.



Jeong-Soo Park received the B.S., M.S., and Ph.D. degrees from the Department of Computer Science, Yonsei University in 2003, 2005, and 2014, respectively. Currently, he is a researcher of the Processor Architecture Lab at Samsung Advanced Institute of Technology. His research interests include 3D graphics hardware, programmable shader architecture, and mobile 3D graphics.

Seok-Yoon Jung received the B.S. and M.S. degrees in control and instrumentation engineering and the Ph.D. degree in electrical engineering from Seoul National University, Seoul, Korea, in 1987, 1989, and 1998, respectively. Since February 1989, he has been a Member of the Research Staff of Samsung Advanced Institute of Technology, Kyungki, Korea. His current research interests include image processing, image and video data compression, and three-dimensional graphics modeling and representation.

Woo-Chan Park received M.S and Ph.D degree in Computer Science, Yonsei University in 1995 and 2000, respectively. Currently, he is a professor at the School of Computer Engineering, Sejong University, Seoul, Korea. His research interests include 3D rendering processor architecture, ray tracing accelerator, parallel rendering, high performance computer architecture, computer arithmetic, and ASIC design.

Dinesh Manocha is currently the Phi Delta Theta/Mason Distinguished Professor of Computer Science at the University of North Carolina at Chapel Hill. He has co-authored more than 380 papers in the leading conferences and journals on computer graphics, robotics, and scientific computing. He has also served program chair for many conferences and editorial boards of many leading journals. Some of the software systems related to collision detection, GPU-based algorithms and geometric computing developed by his group have been downloaded by more than 150,000 users and are widely used in the industry. Manocha has received awards including Alfred P. Sloan Fellowship, NSF Career Award, Office of Naval Research Young Investigator Award, and 14 best paper awards at the leading conferences. He is a Fellow of ACM, AAAS, and IEEE, and received Distinguished Alumni Award from Indian Institute of Technology, Delhi.

Tack-Don Han is a professor in the Department of Computer Science at Yonsei University, Korea. His research interests include high performance computer architecture, media system architecture, and wearable computing. He received Ph.D. in Computer Engineering from the University of Massachusetts.