RACBVHs: Random-Accessible Compressed Bounding Volume Hierarchies

Tae-Joon Kim, Bochang Moon, Duksu Kim, Sung-Eui Yoon, *Member, IEEE*

Abstract—We present a novel compressed bounding volume hierarchy (BVH) representation, *random-accessible compressed bounding volume hierarchies (RACBVHs)*, for various applications requiring random access on BVHs of massive models. Our RACBVH representation is compact and transparently supports random access on the compressed BVHs without decompressing the whole BVH. To support random access on our compressed BVHs, we decompose a BVH into a set of clusters. Each cluster contains consecutive bounding volume (BV) nodes in the original layout of the BVH. Also, each cluster is compressed separately from other clusters and serves as an access point to the RACBVH representation. We provide the general BVH access API to transparently access our RACBVH representation. At runtime, our decompression framework is guaranteed to provide correct BV nodes without decompressing the whole BVH. Also, our method is extended to support parallel random access that can utilize the multi-core CPU architecture. Our method can achieve up to a 12:1 compression ratio and, more importantly, can decompress 4.2M BV nodes (= 135 MB) per second by using a single CPU-core. To highlight the benefits of our approach, we apply our method to two different applications: ray tracing and collision detection. We can improve the runtime performance by more than a factor of 4 as compared to using the uncompressed original data. This improvement is a result of the fast decompression performance and reduced data access time by selectively fetching and decompressing small regions of the compressed BVHs requested by applications.

Index Terms—Hierarchy and BVH compression, random access, cache-coherent layouts, ray tracing, collision detection.

1 Introduction

Bounding volume hierarchies (BVHs) are widely used to accelerate the performance of various geometric and graphics applications. These applications include ray tracing, collision detection, visibility queries, dynamic simulation, and motion planning. These applications typically precompute BVHs of input models and traverse the BVHs at runtime in order to perform intersection or culling tests. Many different types of bounding volume (BV) representations exist such as spheres, axis-aligned bounding boxes (AABBs) and, oriented bounding boxes (OBBs).

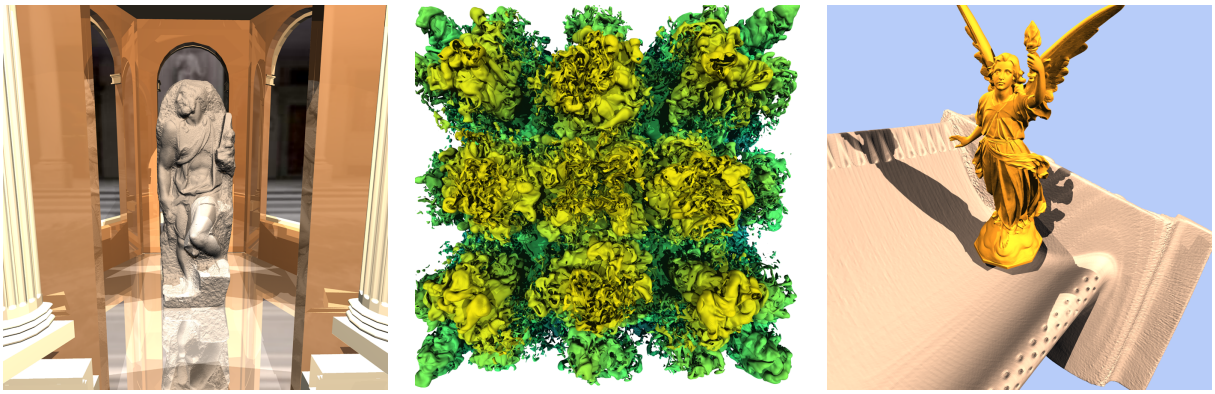
A major problem with using BVHs is that BVHs require large amounts of the memory space. For example, each AABB and OBB node takes 32 and 64 bytes respectively. Therefore, BVHs of large models consisting of hundreds of millions of triangles can take tens of gigabytes of space. Moreover, the typical data access pattern on BVHs cannot be determined at the preprocessing time and is random at runtime. Therefore, accessing BVHs at runtime can have low I/O efficiency and cache utilization.

An aggravating trend is that the growth rate of the data access speed is significantly slower than that of the processing speed [1]. Therefore, the problem of high storage requirements and low I/O efficiency/cache utilization of BVHs will become more pronounced in the future.

Several approaches have been developed to address this problem. One class of methods uses compact in-core BV representations by using quantized BV information or by exploiting the connectivity information of an original mesh and coupling the mesh and the BVH. Another class of methods stores BV nodes in a cache-coherent manner to improve cache utilization and, thus, improve the performance of traversing BVHs. However, due to the widening gap between data access speeds and processing speeds, prior work may not provide enough reduction in storage requirements nor achieve high I/O efficiency during the BVH traversal.

Main results: In this paper, we present a novel compact BVH representation, *random-accessible compressed BVHs (RACBVHs)*, for various applications requiring random access on BVHs of massive models. We present a cluster-based layout-preserving BVH compression and decompression method supporting transparent random access on the compressed BVHs. We compress BVs of a BVH by sequentially accessing BVs in the BV layout of the BVH. During the compression, we decompose consecutive BVs into a set of clusters, each of which will serve as an access point for random access at runtime (Sec. 4). Our compression method preserves the original layout of a BVH and, thus, maintains high-cache utilization during the BVH traversal if the original layout maintains high cache-coherence. In order to allow various applications to transparently access the compressed BVHs, we provide a general BVH access API (Sec. 5). Given a BV node requested by the API, our runtime decompression framework efficiently identifies, fetches, and decompresses a cluster containing the data into an in-core representation that can efficiently support random access. Also, our method is easily extended to support parallel random access that can exploit the widely available multi-core CPU architecture. To demonstrate the benefits of our

All the authors are with KAIST (Korea Advanced Institute of Science and Technology).



(a) St. Matthew scene (b) Iso-surface model (c) Lucy and CAD turbine models

Fig. 1. The left and middle images show results of ray tracing using our random-accessible compressed bounding volume hierarchies (RACBVHs) of St. Matthew model consisting of 128 M triangles and an iso-surface model consisting of 102 M triangles. The right image shows a frame during a rigid-body simulation using collision detection between two models including the Lucy model consisting of 28 M triangles. By using RACBVHs, we can reduce the storage requirement by a factor of 10:1 and, more importantly, improve the performance of ray tracing and collision detection by more than a factor of four over using uncompressed data.

method, we implement two different applications, ray tracing and collision detection, by using our BVH access API (Sec. 6).

Overall, our approach has the following benefits:

1) **Wide applicability:** The provided BVH access API allows various applications to transparently access the compressed BVHs. Moreover, our BVH access API supports random access and does not restrict the access pattern of BVH-based applications. As a result, existing BVH-applications can be easily modified to take advantage of the benefits of our RACBVH representation.
2) **Low storage requirement:** Our RACBVH representation has up to a 12:1 compression ratio compared to an uncompressed BV representation. Also, we use random-accessible compressed meshes (RACMs) and achieve a similar compression ratio for the compressed meshes.
3) **Improved performance:** Our decompression method is fast and processes 4.3 M nodes per second by using a single CPU-core. Also, by selectively fetching and decompressing the small regions of the compressed BVHs and meshes requested by applications, we can reduce expensive data access time. As a result, we can achieve more than a 4:1 performance improvement on our tested applications over using uncompressed BVHs and meshes.

Finally, we analyze our method and provide comparison over prior methods in Sec. 7.

2 Related Work

In this section, we review prior work related to BVHs and compression methods supporting random access on the compressed data.

2.1 Bounding Volume Hierarchies

BVHs have been widely used as an acceleration data structure for ray tracing [2], visibility culling, and proximity computations [3], [4]. There are different types of bounding volumes (BVs). Most commonly used BV types are simple shapes such as spheres [5] and axis-aligned bounding boxes (AABBs) [6], or tighter-fitting BVs such as oriented bounding boxes (OBBs) [7], and discretely oriented polytopes (k-DOPs) [8], etc. Many top-down and bottom-up techniques have been proposed to construct these BVHs [3], [4], [9]. AABBs and spheres are used especially widely due to the simplicity and compactness [4].

2.2 Mesh Compression

In the fields of computer graphics and visualization, mesh compression techniques have been well studied over the last decade and excellent surveys are available [10], [11]. Most previous mesh compression schemes were designed to achieve a maximum compression ratio as they were designed for archival use or for transmission of massive models [12], [13], [14]. They achieved this goal by encoding vertices [12], [15], [16], edges [17], and faces [18], [19], [20] in a particular order agreed upon by the encoder and decoder.

2.3 Compression and Random Access

Most prior mesh compression techniques do not directly provide random access to the compressed meshes. Typically, in order to access a particular mesh element such as vertex, the whole compressed mesh must be sequentially decompressed to an uncompressed format that can support random access. In this section, we

will focus on various compression techniques supporting random access on the compressed data.

Single or multi-resolution mesh compression: Choe et al. [21], [22] proposed a single-resolution mesh compression method that supports selective rendering. Recently, Yoon and Lindstrom [23] proposed random-accessible compressed meshes for general applications requiring random mesh access. This method achieves up to a 20:1 compression ratio and improves the runtime performance of iso-contouring and layout re-computation. Several multi-resolution compression methods also support random access. Gobbetti et al. [24] proposed a compressed adaptive mesh representation of regular grids for terrain rendering. Also, Kim et al. [25] introduced a multi-resolution compression method for selective rendering.

Multimedia and regular grids: Random access is one of the key components of the MPEG video compression format that allows users to browse video in a non-sequential fashion [26]. Particularly, the MPEG video codec method periodically inserts "intra pictures" as access points in the compressed scheme. Such intra pictures are compressed without using information from other frames. Then, subsequent frames are compressed by predicting the motion in between these intra pictures. For regular volumetric grids, wavelet-based compression methods [27], [28] that support random access have been proposed. Also, Lefebvre and Hoppe proposed a perfect spatial hashing method as an efficient random-accessible compressed image format [29].

2.4 Tree and BVH Compression

Tree compression has been studied in many different fields [30]. These techniques include compressing the tree structure by linearizing the structure [31] and transforming the tree into a pre-defined tree [32]. However, these compressed trees do not support random access and do not preserve the layouts of the trees. There are relatively few research efforts on compressing BVHs. A BVH has two main components: BV information and indices to child nodes.

Encoding bounding volumes: In order to compress the bounding volume information, fixed-rate quantization methods [33] are frequently used [34], [35], [36] , as applied to compress geometry of meshes [10]. Also, hierarchical encoding schemes were developed to further achieve a higher compression ratio and improve the compression quality [37], [38] by performing quantization of the bounding volume of a node within the region of the bounding volume of its parent node. These methods can support fast decoding and random access on the quantized bounding volume information. We also employ a hierarchical quantization method. We further improve the compression ratio by using a simple prediction model and dictionary-based compression methods [39] while supporting fast random access.

Encoding tree structures: Many techniques assume a particular tree structure (e.g., complete tree) in order to completely remove any cost related to encoding the tree structure [34], [40]. Recently, Lauterbach et al. [40], [41] introduced a Ray-Strip representation, which implicitly encodes a complete spatial kd-tree from a series of vertices. These techniques do not use any bit for encoding the tree structures. Lefebvre and Hoppe [42] employed local offsets to encode the location of child nodes given the pre-ordered layout of the tree. All of these techniques support random access on the compressed meshes, but assume a particular tree structure or a layout for the tree. Therefore, these methods may not be compatible with various hierarchy construction methods [9], [43], [4] optimized to achieve the high culling efficiency of BVHs.

3 Overview

In this section, we discuss issues that arise when using BVHs of massive models and give a brief overview of our approach to efficiently handle them.

3.1 BVHs of Massive Models

BVHs are widely used to accelerate the performance of intersection or culling tests in various applications. The leaf nodes of a BVH contain triangles of the original model. Each intermediate node of a BVH contains the BV information that encloses all the triangles located under the sub-tree rooted at the intermediate node. In this paper, we use the AABB and a binary BVH due to its simplicity and the wide acceptance in various applications [4], [44].

High storage requirement: The storage requirement of BVHs can be very high for massive models consisting of hundreds of millions of triangles. For example, a simple AABB node representation has the following structure:

```
struct AABB {
  struct BV {
    Vector3f Min, Max;
  }; // bounding volume information.
  struct TreeStructure{
    Index Left, Right;
  }; // indices for child nodes.
};
```

Listing 1. **AABB Node Representation**

The **Min** and **Max** variables store the minimum and maximum extents of the AABB in the x, y, and z dimensions. Also, the **Left** and **Right** variables store the indices of child nodes in the case of intermediate nodes. Typically, a BVH is constructed until each leaf node has only one triangle. For the rest of the paper, we assume that each leaf node of a BVH contains only one triangle and explain our method with this assumption. Later, we extend our method to support multiple triangles in leaf nodes in Sec. 7. In the case that leaf nodes contain only one triangle, the **Left** and **Right** variables of a leaf node

store a triangle index of the triangle and a null index respectively.

This AABB structure requires 32 bytes per node. A model consisting of 100 million triangles requires about 6.4 GB of the main memory. Therefore, BVHs of massive models may not be loaded into the main memory and be accessed from the disk or through the network.

Random access pattern: Traversal on BVHs shows random access pattern for applications including ray tracing and collision detection. These applications typically take two inputs: two 3D objects for collision detection and one 3D object and a ray for ray tracing. The algorithm traverses BVHs of objects in the depth-first or breadth-first order as long as an intersection is detected between two inputs. In general, it is hard to predict the runtime access pattern on BVHs at the preprocessing time or to optimize the access pattern at runtime. The data access time is often the main bottleneck of many applications that use BVHs of massive models.

Cache coherence: There have been several research efforts toward designing cache-coherent algorithms by reordering the runtime access patterns [45], [46] or by reordering the underlying data layout [47], [48]. These techniques reduce the number of expensive cache misses during random access on the data, since cache misses in various memory levels (e.g., L1/L2, main memory, and disk) are significantly expensive compared to the computation time [1].

Out-of-core techniques: Out-of-core techniques have been widely studied in order to handle massive models that cannot fit into the main memory [49]. These techniques aim at reducing expensive data access operations when dealing with massive models by only loading necessary data and performing local operations. However, due to the widening gap between data processing speeds and data access speeds [1], the time spent even on loading only the necessary data from the disk can be very expensive.

3.2 Our Approach

In order to efficiently access BVHs and improve the performance of various applications using BVHs, we propose a novel BVH compression and decompression method supporting random access. Our method has two main components: (1) a cluster-based layout preserving BVH compression and (2) a runtime decompression framework that transparently supports random access on the RACBVH representation without decompressing the whole BVH.

Cluster-based layout preserving BVH compression: We compress BVs of a BVH by sequentially reading BVs in the BV layout of the BVH. We choose our compression method to preserve the original layout of the BVH in order to achieve the high cache utilization which the original layouts may maintain. We decompose the original layout of the BVH into a set of clusters. We assign consecutive BVs in the BV layout to each cluster

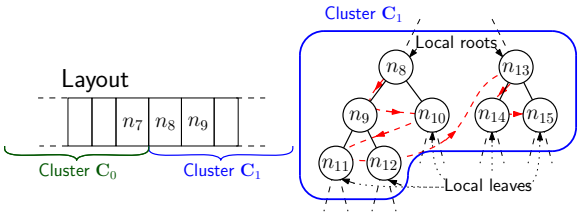


Fig. 2. **Clusters from a BV layout:** This figure shows a BV layout on the left and a computed cluster, C_1, on the right. Red arrows indicate the BV layout of the BVH.

and set each cluster to have the same number (e.g., 4K) of BVs to quickly identify a cluster containing a BV node requested by an application at runtime. We compress each cluster separately from other clusters so that the clusters can be decompressed in any order.

Runtime BVH access framework: We define an atomic BVH access API supporting transparent and random access on the compressed BVHs. Our runtime BVH access framework first identifies a cluster containing a BV node requested by an application. Then, the runtime framework fetches and decompresses the cluster into an in-core representation. Based on our in-core representation, we can very efficiently support random access to applications. Our runtime BVH access framework is guaranteed to return the correct BV information of the requested data when applications access the compressed data via our BVH access API. We employ a simple memory management method based on a least recently used (LRU) replacement policy in order to handle massive models and their BVHs that cannot fit into the main memory.

4 Compression

In this section, we will explain our cluster-based layout preserving BVH compression method.

4.1 Layout Preserving BVH Compression

Our compression method sequentially reads and compresses BV nodes given in the format of Listing 1. As we read each BV node in the BV layout, we assign the BV node into a cluster. For clustering, we simply decompose consecutive BV nodes into a cluster, where each cluster has power-of-two nodes (e.g., 4K nodes). An index of a BV node increases sequentially as we compress each node. Then each BV node index can be encoded as a pair of indices, (C_i, l_i), where C_i is a cluster index to a cluster that the node is assigned to and l_i is a local index of the node within the cluster. Let us call the pair of indices a *pair index*. The pair index is also used in an earlier method of compressing massive meshes [50]. Note that each cluster may have one or multiple sub-trees (see Fig. 2). We call the root nodes of these sub-trees contained in a cluster *local roots* of the cluster. We also define *local leaf nodes* to denote leaf nodes of these sub-trees within each cluster. We define $Parent(n)$ to be a parent node of a node n. Also, *parent clusters* of a cluster c are defined to be clusters containing parent nodes of

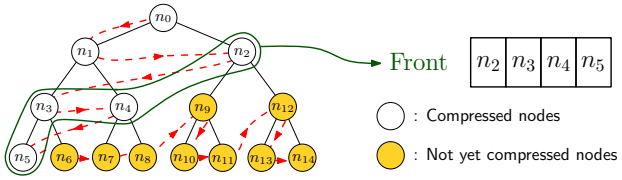


Fig. 3. **A Front during Compression:** This figure shows a front of a sub-tree during compression. We use the front to compactly encode the tree structure of BVHs.

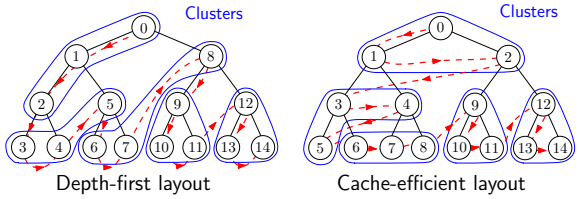


Fig. 4. **Clustering with Different BV Layouts:** This figure illustrates different layouts of a BVH and constructed clusters by assigning the fixed number (e.g., 3) of consecutive BV nodes into clusters.

local roots of the cluster c. Similarly, we can define *child clusters* of a cluster.

For a new cluster, we initialize all the compression contexts. Therefore, each cluster can be decompressed in any order at runtime although we compress each cluster sequentially during the preprocessing time. We also define a *front* for a sub-tree located under each local root of a cluster during compression (see Fig. 3). As we compress each node, we add the node to the front and connect the node to its parent node, if the parent node is in the front. Once a node in the front is connected to its two child nodes, the node is deleted from the front. As a result, the front consists of BV nodes that are not yet connected to its two child BV nodes during compressing the cluster. We will use the front in order to compactly encode the tree structures of BVHs.

It is desirable that constructed clusters should contain BV nodes that are likely to be accessed together during the traversal of BVHs. Otherwise, we may have to load and decompress many clusters, which could lower the performance of applications. Since clusters are implicitly computed from original layouts of BVHs, BV nodes that are likely to be accessed together should be stored very closely in the layout of a BVH. Fortunately, there are a few layouts that satisfy this property. These layouts include the cache-efficient layouts of BVHs [48] and the van Emde Boas layout [47]. An example of clusters computed from different layouts is shown in Fig. 4. We will compare the performance of different layouts in Sec. 7.

We also propose using a dictionary-based compressor and decompressor [39] to achieve a high compression ratio and, more importantly, a fast decompression performance for efficient random access on our RACBVH representation. A pseudo-code of our compression method is given in Algorithm 1, for the clarity of explaining our method.

4.2 Encoding Bounding Volumes

The **Min** and **Max** variables of the BV node representation (Listing 1) store the minimum and maximum extents of an AABB node. We first quantize each component, i.e., x, y, and z, of the **Min** and **Max** of each BV node. We use a fixed-rate quantized method [33] for those components based on the root bounding volume of a BVH. We quantize the components conservatively to ensure that the quantized BV still encloses all the triangles of the original BV. Then we further compress the quantized **Min** and **Max** values based on the BV information of $Parent(n)$ of the node n currently being encoded. To do that, we predict two child BVs of $Parent(n)$ given the BV of $Parent(n)$. Then we only encode the difference between the predicted and actual BVs using our dictionary-based encoder. For the prediction, we partition the parent BV into two child BVs by dividing the longest axis of the BV into half. We find that this simple *median prediction method* works well in our tested benchmarks.

It is possible that a cluster containing a parent node $Parent(n)$ of a node n may be different from a cluster containing the node n. In this case, we cannot assume that the BV information of the $Parent(n)$ is available when decompressing the node n. Our runtime framework addresses this problem by guaranteeing the existence of the $Parent(n)$ by loading the cluster containing the $Parent(n)$. This will be discussed more in Sec. 5.

4.3 Encoding Tree Structures

The **Left** and **Right** of intermediate BV nodes (Listing 1) contain two child indices and represent the tree structure. However, the **Left** of a leaf node has a triangle index contained in the leaf node. We will describe our compression method for child indices of intermediate nodes and, next, for triangle indices of leaf nodes in the next two sections.

4.3.1 Encoding Child Indices

Initially, we attempted to directly encode child indices of the **Left** and **Right**. However, we found this approach shows poor compression results. This is mainly caused by the fact that there are many factors (e.g., layout types and tree structures) affecting child indices of a node and it is difficult to account for these factors when predicting child indices. Instead of encoding child indices given a node, we propose to encode a parent node index from a node n currently being encoded. The main rationale behind this design choice is that the tree structure above the node n is available at the time of encoding or decoding the node n. Therefore, encoding a parent node index is simply choosing a node from the already encoded tree structure instead of predicting a tree structure below the node n.

Note that either the **Left** or **Right** index of a parent node $Parent(n)$ is equal to the node index n_{idx} of the node n. Therefore, if we encode a parent node index of the node n, then, we can access its parent node $Parent(n)$ and fill either the **Left** or **Right** indices of

$Parent(n)$ with the node index n_{idx} when we decompress the node n. To indicate whether the node n is the left or right child of its parent, we encode an additional bit.

Our BV node does not directly store any information about its parent node index. Fortunately, this information can easily be constructed during compression. To do this, we maintain a hash table. For a node n specified by a node index n_{idx}, our BV representation gives us indices of two child nodes, **Left** and **Right**, of the node n. We simply construct two hash map elements connecting a key of each child index to n_{idx} of the node n.

As we encode each node n, we attempt to find its parent node index by querying its node index n_{idx} to the hash table. If we can find the node index n_{idx} and, thus, its parent node index from the hash table, then we can find the parent node $Parent(n)$ specified by the parent node index among nodes stored in the front according to the definition of the front described in Sec. 4.1. For example, when we encode a node of n_6 in the example of Fig. 3, we first attempt to find its parent node n_3 using the hash table by querying the node n_6. Since the hash map element that connects n_6 to n_3, is already inserted when n_3 is encoded, we can find its parent node n_3. Also, the node n_3 is the second element in the front. Therefore, we encode 2. Then we insert n_6 to the front and remove n_3 from the front since the node n_3 is connected to its two child nodes. Finally, we add two hash map elements that connect two child nodes of n_6 to the node n_6.

Note that the number of nodes stored in the front is typically much smaller than the number of nodes in the BVH. Therefore, we can compactly encode the parent node index by encoding its position in the front. The average size of the front with the cache-efficient layouts is 13 when we assign 4 K nodes for each cluster in our tested models. We can compute the position of the parent node in the front and update the front in a constant time by using a dynamic vector for nodes in the front.

If the node n currently being encoded is the root node of the BVH or a local root of a cluster that the node n is assigned to, then we cannot find its parent node index by querying a node index n_{idx} to the hash table. The case of the root node can easily be identified and addressed. In the case of local roots of a cluster, the parent nodes of those local roots are located in another cluster. Therefore, the front of the cluster that is currently being encoded does not have any information about the parent nodes of the local roots while the cluster is being compressed. One naive solution is to directly store the parent node index, which requires storing many bits. Instead, we encode the parent node index by decomposing it into a pair index of (C_i, l_i). Then we encode C_i among the parent clusters of the cluster that is currently being encoded and, then, encode l_i. We found that this method gives a high compression ratio since the number of parent clusters is typically very small (e.g., 2 on average for cache-efficient layouts).

Local leaf nodes: Given our compression scheme, we cannot compute the child indices of local leaf nodes of a cluster without decompressing its child clusters at runtime. This is because these child indices of local leaf nodes will be computed when processing child clusters containing the child nodes specified by the child indices. However, loading the child clusters of a cluster at runtime can significantly increase the working set size of applications and, thus, decrease the performance of applications. To avoid this problem, we directly encode left and right child indices stored in **Left** and **Right** only for local leaf nodes. To compactly encode left and right child indices of local leaf nodes, we employ a simple delta encoding. Given a child index that we are going to encode, we compute the difference between the index and a previously encoded index. Then we encode the difference using our dictionary-based encoder.

Algorithm 1 Encode(N)

```
Require: A node index N
  Index P    // Parent node index of N
  if N ∉ Hash, the hash table then
    // N is a local root
    Encode(0)
    Assign and encode N's parent node to P
  else
    // N is not a local root
    P ← Hash.Find(N)
    int k ← Front.GetOffset(P)
    Encode(k + 1)
  end if

  Update Front and Hash.

  // Quantize and encode the BV of the N
  QBV pQBV ← Quantize(GetBV(P))
  QBV nQBV ← Quantize(GetBV(N))
  QBV predQBV ← Predict(pQBV)
  Encode( diff. between predQBV and nQBV)

  if N is a leaf node then
    // Encode triangle indices
    Index triIdx ← GetTriIdx(N)
    Encode(triIdx − lastTriIdx)
    lastTriIdx ← triIdx
  else if N is a local leaf then
    // Encode child indices
    Index leftIdx ← GetLeftChildIdx(N)
    Index rightIdx ← GetRightChildIdx(N)
    Encode(leftIdx − lastChildIdx)
    Encode(rightIdx − leftIdx)
    lastChildIdx ← rightIdx
  end if
```

4.3.2 Encoding Triangle Indices

The **Left** index of a leaf node contains a triangle index. Since a BVH is constructed by spatially partitioning the triangles of a mesh, triangles stored in neighboring leaf nodes are highly likely to be spatially close and may even share edges between them. We use this observation to design two different compression methods that explicitly or implicitly exploit the connectivity of a mesh between leaf nodes.

Explicit utilization of the mesh connectivity: Our first method explicitly uses the underlying mesh connectivity. We encode triangle indices when we encounter leaf

nodes while sequentially accessing BV nodes in the BV layout. For each encoded triangle index of a triangle, we compute three indices of its three neighboring triangles and store them in a cache. Then for a triangle index of a next leaf node, we attempt to encode the triangle index among three triangle indices stored in the cache. Otherwise, we encode the triangle index. We found that this method works well and about half of triangles indices of leaf nodes can be encoded by the small cache holding three previous neighboring triangle indices. However, one downside of this approach is that, to compute each triangle's neighbors, the mesh connectivity must be constructed during decompression, which is inefficient at runtime.

Implicit utilization of the mesh connectivity: In order to support efficient decompression and high compression ratio, we propose to use a simple delta coding method. We found that the difference between the previous and the current triangle indices is typically small when layouts of a BVH and a mesh have high coherence. Therefore, we encode the triangle index difference with our dictionary-based encoder if the difference is less than a small threshold (e.g., 10). We found that about 80% of the differences are within the threshold of 10 when using cache-efficient layouts. Otherwise, we encode the triangle index. This compression method implicitly utilizes the underlying layouts of both BVHs and meshes having the high spatial coherence between neighboring nodes and triangles. As a result, this implicit method requires fewer bits per node (bpn) and decompresses much more quickly.

Meta file: In order to support random access on the compressed BVH at runtime, we construct a meta file as we compress a BVH. The meta file has a starting address of each cluster in the compressed BVH and the number of BV nodes assigned to each cluster. Since the meta file takes only minor memory space (e.g., less than one MB), we store the meta file without any compression. Note that this meta file is constructed progressively as we compress a BVH in one pass.

4.4 Dictionary-based Compression

We employ one more layer of compressing the data by using a dictionary-based compressor to improve the compression ratio while achieving high decompression performance. Particularly, we use variations of the LZW method [39, pages 199–208] based on a simple dictionary. We choose the LZW method since it can quickly detect and compress repeating patterns in the sequences of symbols. For each different compression context, we initialize dictionary entries with all the symbols that the compression context can have. Then we allow adding a new entry consisting of a combination of symbols if the entry is not in the dictionary.

For the compression contexts of the delta encoding and the difference encoding for the quantized BVs, we further optimize our LZW method since these compression contexts show different characteristics compared to other contexts. In these compression contexts, we do not initialize the dictionary with symbols and start with an empty dictionary since there are too many possible symbols and we found that most of those symbols do not appear during compressing a cluster. We add a new symbol if we encounter the symbol during compression. However, we do not add new entries consisting of a combination of symbols since we found that there are not many repeating patterns in these compression contexts.

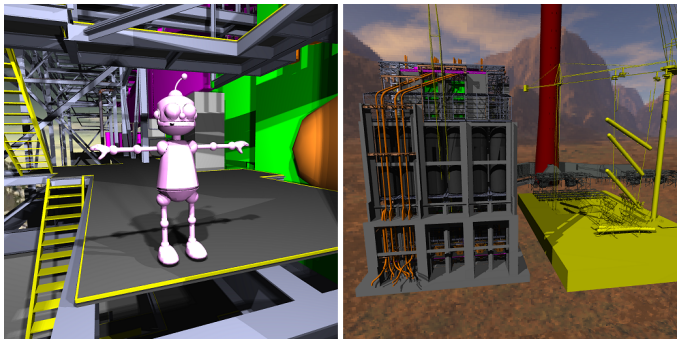


Fig. 5. **Hubo and Power Plant Models:** The Hubo robot model (16 K triangles) is placed in the upper left corner of the power plant model (13 M triangles). The entire power plant model is shown on the right. We improve the performance by a factor of 2:1 for collision detection using our RACBVH representation.

4.5 Random-Accessible Compressed Meshes (RACMs)

We employ the RACM representation [23] to further reduce the storage requirement of meshes, which are used together with BVHs for various applications. We use OpenRACM library [51] available online for computing RACMs and employ the provided mesh access API to access the compressed meshes. The original RACM method used the arithmetic encoder [52] to achieve a high compression ratio. We found that our dictionary-based encoder shows much higher runtime performance than the arithmetic encoder. Therefore, we modify the RACM method to use our dictionary-based method. We will provide more detail comparisons between our dictionary-based encoder and the arithmetic encoder in Sec. 7.2.

5 Runtime Decompression Framework

In this section, we present our runtime decompression framework that transparently supports random access on our RACBVH representation.

In-core BV representation: As we decompress BV nodes requested by applications, we store the decompressed BVs in the main memory in the format of the simple BV representation shown in Listing 1. The main benefit of using this in-core BV representation is that it can directly support random access without any computation overhead.

5.1 BVH Access API

In order to allow various applications to transparently access our RACBVHs, we provide the following BVH access functions:

Index **GetRootIndex** (void): Return an index of the root node of a BVH.
BV & **GetBV** (Index n_{idx}): Return the BV information specified by the node index n_{idx}.
bool **IsLeaf** (Index n_{idx}): Return whether a BV node specified by the node index n_{idx} is a leaf.
Index **GetLeftChildIdx** (Index n_{idx}): Return an index of the left child node of a BV node specified by the node index n_{idx}. We also define a similar function for the right child.
Index **GetTriangleIdx** (Index n_{idx}): Return a triangle index stored at the leaf node specified by the node index n_{idx}.

Based on this BVH access API, we can traverse a BVH in a hierarchical manner or access any BV nodes in an arbitrary order. Also, we can support more advanced BVH access methods like front-based traversal [53] or hierarchy traversal from an arbitrary entry point [54], based on these atomic BVH access API.

5.2 Runtime BVH Access Framework

Our runtime data access framework first reads the meta file. Suppose that an application requests a BV node by calling the **GetBV** (·) function. Then, the runtime data access framework identifies a cluster containing the requested data, decompresses it into our in-core BV representation, and returns the data to the application. Since clusters have consecutive power-of-two BV nodes, we can compute a cluster index by performing a few bit operations to a given node index. Once a cluster index is computed, we refer to the meta file to acquire a starting address of the cluster on the compressed BVH and, then, we decompress the cluster.

Decompressing a cluster: The process of decompressing a cluster is symmetric to the process of compressing it. As we read the compressed data of a cluster, we reconstruct the tree structures represented by the **Left** and **Right** indices and the BV information. The tree structure is reconstructed without extracting any information from other clusters. On the other hand, the BV information of a node is reconstructed by extracting the BV information from its parent node given our hierarchical BV compression method. For the local roots of a cluster, we cannot reconstruct the BV information of these local roots and their sub-trees if the BV information of parent nodes of these local roots is not available while decompressing the cluster. We call these nodes that we cannot reconstruct the BV information at the time of decompressing the cluster *incomplete nodes*. We also call all the rest of the nodes *complete nodes*. We use the most significant bit of **Right** to indicate whether a node is complete or not at runtime. For complete nodes, we can reconstruct their BV information as we decompress a cluster since their parent BV nodes are available. For each incomplete node, we decompress the differences between the predicted and actual BVs and, then, store them at the **Min** and **Max** variables. Also, we store the decompressed parent index at **Left** instead of computing the left and right indices of the node and storing them at **Left** and **Right**. We lazily construct the BV information and tree structures of incomplete nodes based on the data stored in the in-core BV representation.

BV completion: If the BV node requested by **GetBV**(·) is complete, we can simply return the stored in-core BV representation to the application. If the node is incomplete, then we search for its local root, n_l. This operation can easily be performed by using the parent node index stored in the **Left** of the incomplete node n. Then, we force to load a cluster that the parent node of the local root n_l is assigned to, if the cluster is not yet loaded. Note that, in order to load a cluster for the BV completion, we may need to recursively load another cluster. Fortunately, during the typical hierarchical traversal of a BVH, a cluster would have been already loaded if a node located in the cluster's subtree has already been accessed. Once we obtain the BV information of a parent node of the local root n_l, we can reconstruct the BV information based on the BV difference stored at the **Min** and **Max** of these incomplete nodes. We also compute the left and right child indices for the **Left** and **Right**. We complete the BV information of all the incomplete nodes of the sub-tree rooted at the local root node n_l within the cluster that n_l is assigned to. Then we set those incomplete nodes to be the complete nodes.

We explained how our runtime framework handles a call of **GetBV** (·). Our runtime framework also performs the similar procedure for all the other functions except for **GetRootIndex** (), which is processed very easily. Note that all the BVH access functions except for **GetRootIndex** () are called with a parameter of a node index, n_{idx}. These functions first find the requested node in the same manner of processing **GetBV** (·). Once the node is identified, we simply return the requested information (e.g., BV, child index, or triangle index) of the node to the application.

5.3 Memory Management

In order to handle massive models whose BV nodes and meshes cannot fit into the main memory, we employ a simple memory management. Given a pre-allocated memory pool of a size specified by the user, we perform the memory management at the granularity of clusters. We maintain a LRU-list of clusters that have been accessed by our BVH access API. Since updating the LRU-list of clusters is expensive, we update the list only when we encounter a new cluster that is different from the previously accessed cluster. Note that clusters located in lower BVHs are accessed less frequently than clusters located in upper portions of BVHs during the BVH traversal. Our simple LRU-based replacement method implicitly considers this factor since clusters located at upper portions of BVHs are more likely to be re-visited during the BVH traversals and, thus, they are less likely to be unloaded.

Pre-loading the RACBVHs: Our RACBVH representations associated with the RACM representations require

TABLE 1
Benchmark Models and Compression Results

Model	Tri. (M)	Vert. (M)	Size (MB) of uncompressed BVH	Size (MB) of uncompressed Mesh	Size (MB) of compressed BVH	Size (MB) of compressed Mesh	Compressed bpn Tree	Compressed bpn BV	Compressed ratio BVH	Compressed ratio Mesh
St.Matthew	128	64	7811	3933	814	320	8.66	18.0	9.6:1	15.4:1
Iso surface	102	51	6254	3912	709	288	8.75	20.3	8.8:1	17.0:1
Lucy	28	14	1684	1008	193	73	8.70	20.7	8.7:1	17.3:1
Power plant	13	11	778	389	66	55	9.22	12.3	12:1	8.8:1
CAD turbine	1.8	0.9	108	67	13	5.5	9.19	21.6	8.3:1	15.2:1

Model complexity, compression results, and compression ratios for our benchmark models are shown. ***Tree*** *indicates tree structures in BVHs and* ***BV*** *indicates the BV information. Our method achieves up to a 12:1 compression ratio over the uncompressed BV representation shown in Listing 1. We use 16 bits for the quantization of the BV information, 4 K node cluster size, and cache-efficient layouts.* ***bpn*** *stands for bits per node.*

much less storage than uncompressed BVH and mesh representations. Therefore, it is possible to sequentially pre-load all the compressed data of massive models into 2–4 GB sized main memory of commodity hardware and access those data without the expensive disk I/O access at runtime. Since the sequential access to the disk during the pre-loading is much faster than random access [1], the pre-loading can be done quickly. We will show the performance of tested benchmark applications with and without pre-loading the compressed data in Sec. 6.2.

6 Results

We have implemented our compression method, out-of-core runtime decompression framework, and benchmark applications on a 3.0 GHz Intel Core2 Extreme-PC that has a quad-core CPU, with 32-bit WindowsXP, 4 GB of RAM, and a SATA disk drive having a sequential reading performance of 62 MB per second. We perform various tests by using a single thread, unless mentioned otherwise. We set our runtime decompression framework to use no more than 2.3 GB of the main memory to cache uncompressed data. Our compression method works with any layouts or BVH construction methods.

Benchmark models: We have tested our method with various benchmark models including the St. Matthew model (128 M triangles, Fig. 1(a)), the Lucy model (28 M triangles, Fig. 1(c)), a CAD turbine model (1.8 M triangles), the power plant model (13 M triangles, Fig. 5), a Hubo robot model (16 K triangles), and an iso-surface model (102 M triangles, Fig. 1(b)) extracted from a scientific simulation. More detail information about our benchmark models is shown in Table 1.

6.1 Compression Results

We construct BVHs of benchmark models and store them in a cache-coherent manner, particularly, cache-efficient layouts [48]. We choose this layout since it shows a high compression ratio and, more importantly, the best runtime performance among the tested layouts, as we will see later. We quantize the BV information using 16 bits. In this configuration, we are able to achieve 27.5 bits per node (bpn) on average for our benchmarks. For the St. Matthew model, our compression method spends 18.0 bpn to encode the BV information and 8.66 bpn to encode tree structures. Compared to the uncompressed AABB representation (Listing 1), we achieve up to a 12:1 compression ratio in our benchmark models. In addition, we achieve about a 13:1 compression ratio for the tested meshes by using RACMs. Overall, we achieve a 10:1 compression ratio on average by using RACM and RACBVH representations compared to using uncompressed BVHs and meshes. Please refer to Table 1 for more detail compression results.

Decompression performance: Our compression method can compress 0.4 M nodes per second. On the other hand, our decompression method can process 4.2 M nodes (= 135 MB) per second when we decompress clusters sequentially. The one order of magnitude faster decompression performance is mainly because we do not need to construct various data structures like the hash table that are only needed during the compression process. Because of the fast decompression performance, our method can improve the performance of many applications that use BVHs of massive models.

6.2 Benchmark Applications

We implement two different applications, ray tracing and collision detection, to verify the benefits of our proposed method. We choose these two applications because they have different access patterns. Ray tracing typically traverses larger portions of BVHs while collision detection accesses smaller and more localized portions of BVHs. We implement these two applications with the proposed BVH access API. For comparison, we can also set our BVH access API to access uncompressed BVHs stored in the format of our in-core BV representation on the disk without changing any application code. Note that applications get the original un-quantized BV information from the stored uncompressed BV nodes. Therefore, applications get tighter BV information, which can achieve higher culling efficiency during the BVH traversal, compared with our quantized BVs of the RACBVHs.

6.2.1 Ray Tracing

We implement a BVH-based ray tracer [44], [55] for the distributed ray tracing [56]. We construct BVHs optimized with the surface-area heuristic (SAH) [9], which is a well known method for constructing acceleration hierarchies that maximize the performance of ray tracing. We also use the projection method [9] for fast triangle-ray intersection tests. To do this, we compute various quantities (e.g., best projection planes, triangle normals, etc.) on the fly as we read and decompress the RACM representation. We use 512 by 512 image resolution for the image generation tests.

We first test our BVH-based ray tracer only using primary rays with small models (e.g., the Stanford bunny model). Our single-thread BVH-based ray tracer can process 1 million rays per second. Note that since our ray tracer handles massive models that cannot fit into the main memory, it runs at an out-of-core mode, which

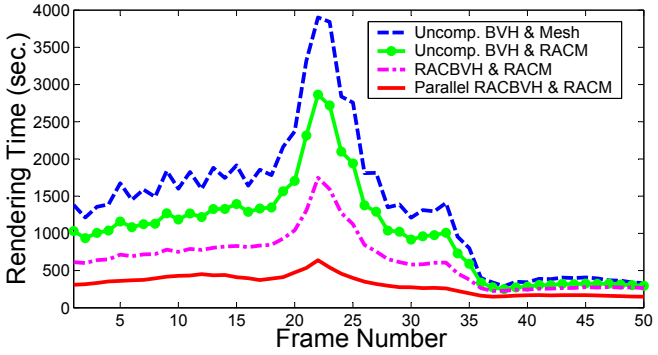


Fig. 6. **Performance of Ray Tracing St. Matthew Scene:** This graph shows the rendering time during ray tracing the St. Matthew scene shown in Fig. 1(a). By using random-accessible compressed meshes (RACMs) without the random-accessible compressed BVHs (RACBVHs), we achieve a 1.5:1 performance improvement over using the original uncompressed data. By using both RACBVHs and RACMs, we achieve a 2.2:1 runtime performance improvement on average. Also, by using four threads with RACBVHs and RACMs (labeled as Parallel RACBVH and RACM), we achieve an additional 2 times improvement and thus improve the performance by a factor of 4.4:1 over using the original uncompressed data. Accessing uncompressed data with multiple threads does not improve the performance because the disk I/O access is the main bottleneck.

requires another data management layer for accessing massive data. We expect that the ray-packet methods [9] can further improve the performance of our ray tracer.

We test our method with one light source and no reflection when we look at St. Matthew model as shown in Fig. 1(a). In this configuration, many of the rays are coherent. By only using RACMs with the uncompressed BVHs, we achieve a 1.5:1 performance improvement over using the original uncompressed data. However, by using both RACBVHs and RACMs, we are able to achieve a 2.6:1 performance improvement on average over using the uncompressed data. We also test the pre-loading of the compressed data. In this case, we spend 18 seconds on sequentially reading and pre-loading 1.1 GB of the RACM and RACBVH. We observe a 4.4:1 performance improvement over the original uncompressed data at runtime. Note that we cannot pre-load the uncompressed data since it takes about 11.7 GB.

We also measure the rendering time during ray tracing the St. Matthew model in the scene setting shown in Fig. 1(a). In this scene, we use 7 point light sources, reflections, and 3 by 3 stratified sampling; this configuration results in many incoherent rays. Initially, the camera is located far from the St. Matthew model and, then, it zooms to the face of the St. Matthew; please refer to the accompanying video. We observe a 1.5:1 and a 2.2:1 performance improvement on average by using RACMs with uncompressed BVHs and by using both RACMs and RACBVHs respectively over by using the original uncompressed data. Fig. 6 shows the ray tracing time for each of these three methods. The performance improves by using RACBVHs and RACMs because the I/O time is reduced due to the fast decompression performance and selective decompression during the BVH traversal.

We also measure the performance of ray tracing the iso-surface model shown in Fig. 1(b). We use 5 point light sources and 3 by 3 stratified sampling. In this configuration, we use both RACM and RACBVH representations. We improve the performances by a factor of 2.6:1 and 1.9:1 on average over using uncompressed data with and without the pre-loading the compressed data respectively.

6.2.2 *Collision Detection*

We implement collision detection and integrate it into a rigid-body simulation [57]. Collision detection identifies colliding regions between two models by traversing BVHs. We create a benchmark where we drop the Lucy model on top of the CAD turbine model (see Fig. 1(c)).

We measure the collision detection time during a simulation that runs for 215 simulation steps. In the tested benchmark models, the sizes of compressed RACBVHs and RACMs are 198 MB and 78 MB compared to 2.8 GB of the original meshes and BVHs. Since RACMs and RACBVHs can be stored in main memory, we pre-load RACMs and RACBVHs, if they are used. Pre-loading RACMs and RACBVHs take about 4 seconds. We improve the performance of the collision detection on average by a factor of 1.4:1, 2.5:1, and 2.9:1 using only RACMs, only RACBVHs, and both RACMs and RACBVHs respectively over using uncompressed data. On average, collision detection accessing RACMs and RACBVHs takes 5 milliseconds (ms) per step. We also test and measure performance of another collision detection benchmark consisting of the power plant and Hubo models (see Fig. 5). In this case, we achieve a 2.1:1 performance improvement over using uncompressed data.

6.3 Parallel Random Access

Our compression and decompression methods can be extended to support parallel random access and exploit the widely available multi-core CPU architecture. To enable the parallel random access in our method, we use a lock to avoid completing the same node from multiple threads. Also, we use a pseudo LRU method, the second-chance algorithm [58] that reduces locking the same node in the LRU list during the update of the list. We test the performance of our method with the ray tracing benchmark. For our parallel ray tracer, we simply divide the image plane into multiple units and assign each unit to a thread. Then, each thread generates and processes primary and secondary rays while accessing compressed data. The memory pool that caches decompressed data is shared among threads. We use four threads and pre-load all the compressed data. We measure the rendering time during the ray tracing of the St. Matthew model with incoherent rays in the scene setting described in Sec. 6.2.1.

When we use the original data, the performance decreases by a factor of 1:1.8 by using four threads over using a single thread. Since four different threads request more disk I/O accesses, these more I/O accesses worsen the disk seek performance and lower the performance of ray tracing. On the other hand, our method achieves

TABLE 2
Cluster Size vs. Performance

Cluster size	256	512	1K	2K	4K	8K	16K
Compression ratio	9.3	9.7:1	9.8:1	9.7:1	9.6:1	9.4:1	9.2:1
RT w/ coherent rays (s)	352	339	351	329	319	318	316
RT w/ incoherent rays (s)	2,220	2,242	2,278	2,307	2,339	2,422	2,470
Collision detection (ms)	4.69	4.74	4.93	5.02	5.34	5.76	6.13

This table shows the compression ratio and ray tracing time (RT) of the St. Matthew model and collision detection time, as a function of the cluster size.

a 2.1:1 performance improvement over using a single thread. Therefore, our method achieves a 4.4:1 performance improvement for ray tracing the St. Matthew model over using the original data. This higher scalability of our method is achieved by removing the expensive and low-performing disk I/O access. The performance of ray tracing the St. Matthew model when using four threads is shown in Fig. 6.

7 Analysis and Comparison

In this section, we analyze the performance of our method and compare its performance with other related techniques.

7.1 Analysis

The performance of our method is affected by several factors. We discuss them in terms of their impact on the runtime performance of applications. We report various results by using a single CPU core.

Cluster size: In order to see how the compression ratio varies as a function of cluster sizes, we compute different versions of the RACBVHs of the St. Mathew model with different cluster sizes ranging from small to large sizes: 0.5K, 1K, 2K, 4K, 8K and 16K nodes (see Table 2). Note that encoding BVs and parent nodes in the front requires more bits as the cluster size becomes larger, since BV prediction errors vary more and the front's size is larger. On the other hand, encoding parent nodes of local root nodes and child indices of local leaf nodes require fewer bits as the cluster size becomes larger, since there are fewer local roots and leaf nodes. Given these two factors, we achieve the highest compression ratios when the cluster size is 1 K nodes. However, the compression ratios are rather stable in the tested cluster sizes.

We also test the performance of ray tracing and collision detection with the different cluster sizes. We first measure ray tracing time of St. Matthew model with coherent rays. The performance improves even when we use very large clusters, since the disk I/O performance improves by reading larger clusters and loaded clusters may be reused during the processing of many coherent rays. However, when we perform ray tracing with incoherent rays, the performance worsens as the cluster sizes become larger, since we have to frequently load and unload clusters, caused by incoherent data access pattern. Also, the performance of collision detection goes down as the cluster size is larger, since we have to load larger clusters, where most of nodes may not be accessed given the very localized data access pattern of collision detection. Although it is very hard to conclude which cluster size is the best across different applications, we found that clusters with 1 K to 4 K nodes show high performances among the tested applications.

Layouts: We also compare the performance of applications with different layouts of BVHs. We compute depth-first, van Emde Boas [47], and cache-efficient layouts [48] of our tested benchmark models. Surprisingly, the best compression results are achieved with the depth-first layouts. However, we found that cache-efficient layouts show the best runtime performance, followed by van Emde Boas, and depth-first layouts. This is mainly because of the high spatial and cache coherences that the cache-efficient layouts maintain. Performance results with different layouts of ray tracing the St. Matthew model are shown in Table. 3.

Number of triangles per leaf node: Our simple BV representation (Listing 1) can be extended to store multiple triangles by using a global triangle index list. If a leaf node contains multiple triangles, we store indices of these triangles consecutively in the list. Then the **Left** and **Right** simply contain the starting and ending positions of these triangle indices in the list. For efficient intersection tests, we compute a sub-BVH for triangles contained in a leaf node on the fly. Also, we cache the computed sub-BVHs of leaf nodes and apply our memory management method to these cached data. We compare the performance of ray tracing the St. Matthew model with different number of triangles per leaf node: 1, 4, 16, and 128. As the number of triangles per leaf node increases, the size of BVHs decreases. The performance peaks when we assign 16 triangles to each leaf node and, in this case, our method shows a 1.8:1 performance improvement over using the uncompressed data.

Extensions to other types of BVs and k-ary BVHs: At a high level, our compression method compresses a BV by first quantizing the BV and predicting its two child BVs by assuming a most likely partitioning plane during the BVH construction. This idea can be applied easily to other types of BVs such as spheres and OBBs, since these BVs and AABBs are constructed using a similar hierarchical partitioning scheme. Also, our method could be extended to support k-ary tree structures. During encoding child indices, we explicitly encode extra information to indicate whether a node is the left or right node of its parent node. In the case of k-ary tree structures, we can simply encode a position of a node among k different child nodes.

Limitations: Accessing the RACBVH and RACM representations has some overhead. For example, we have to perform a few bit operations computing cluster indices for each **GetBV (·)** call. Also, the compressed BVs may give fewer tight extents and cause more intersection tests for applications because they are quantized more conservatively than the original uncompressed BVs. Therefore, when all the data needed to perform an application are already in main memory, our method might lower the

TABLE 3
Layouts vs. Performance

Layouts	Size of RACBVHs(MB)	Compression ratio	Rendering time(sec.)
Depth-first	776	10.1:1	334
van Emde Boas	777	10.0:1	332
Cache-efficient	814	9.6:1	319

This table shows the size of our RACBVHs with different layouts. The compression ratio is computed over the uncompressed in-core BV representation. Cache-efficient layout shows the best runtime performance during ray tracing the St. Matthew model.

performance. To verify this, we perform the ray tracing of a simplified St. Matthew model consisting of 8 M triangles, whose BVH and mesh can be stored in main memory. We load all the data into main memory. In this case, we observe 1% more intersection tests and, thus, about a 1% lower performance using our method than using the uncompressed data at runtime. Also, we perform the collision detection of the rigid-body simulation by using simplified models that can be stored in main memory. We observe 3% more intersection tests and 3% lower performance using our method than using the uncompressed data.

7.2 Comparison

We compare our method with prior work on reducing sizes of BVHs.

Hierarchical quantization methods: QSplat [37] and Quantized kd-tree [38] employed fixed-rate quantization methods. We also use a fixed-rate quantization method and further compress them by using a simple prediction method and encoding prediction errors with a dictionary-based encoder. We choose to use this more aggressive compression method to further reduce the storage requirement and lower the expensive data access time, which is getting more expensive given the current computation trend [1]. We compare our compressed representation over the quantized BVH representation, whose bounding volume information is quantized to a fixed 16 bits. Our method achieves a 6:1 compression ratio over the quantized BVH representation. Moreover, our method still improves the runtime performance 2.3 times over using the quantized BVH for ray tracing the St. Matthew model.

Gzip compression: One can use the gzip compression method to compute compact BVHs. We compare our method with the gzip compression method. Since BVHs that are compressed by gzip do not provide random access directly, we perform gzip compression for each cluster. We use the zlib library [59]. Gzip achieves a 3.2:1 compression ratio over the original uncompressed BVHs and its decompression throughput is 17 MB per second. However, it performs 6.7% slower than using uncompressed data for the ray tracing, mainly because of its low decompression performance. Our method achieves a compression ratio about 3 times higher and a decompression performance six times faster. As a result,

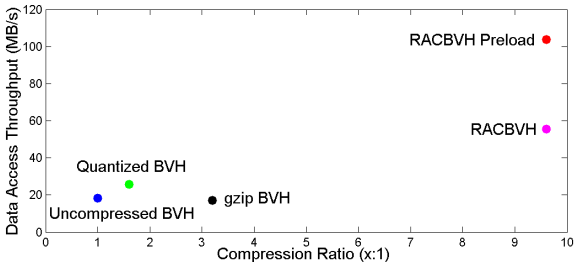


Fig. 7. **Data Access Throughput vs. Compression Ratio:** This graph shows the data access throughput during ray tracing and the compression ratio of each compression method. The data access throughput is computed by summing the data accessing time and decompression time. Uncompressed BVHs do not require any decompression time. We set the data fetching time to be zero in the case of pre-loading the RACBVHs. Note that the higher compression ratio does not always cause the higher data access throughput.

our method improves the runtime performance more than 3 times over using gzip for the ray tracing. This result is not surprising since gzip uses a combination of LZ77 and Huffman encoding [39], which are expensive compression methods which do not work well for the non-repeating floating values found in the original uncompressed BVHs. We measure the data access time including the I/O time and decompression time during ray tracing the St. Matthew model with different compression methods and original uncompressed data (see Fig. 7). As can be seen in the figure, our RACBVH representation shows much higher compression ratio and lower data access time.

Ray-Strip methods: Recently, Lauterbach et al. [40] proposed a *RayStrip*, an in-core compact hierarchy representation for ray tracing and further improved it in a following work [41]. Its main idea is to compute a triangle strip and build a balanced implicit spatial kd-tree on the triangle strip. This method is optimized for an in-core ray tracer and is not tested with other applications. Since this method always uses a complete spatial kd-tree, its runtime performance may be lower than ray tracers that use optimized hierarchies (e.g., hierarchies optimized with the SAH). Also, our method achieves a 50% higher compression ratio and may perform better because it can use optimized BVHs. However, the Ray-Strip representation can be used as a compact in-core representation in our runtime framework for the improved runtime performance. The Ray-Strip representations could be used for lower regions of BVHs whose hierarchy quality has less impact on the overall ray tracing performance.

Statistical compression methods: Most prior compression methods that target higher compression ratio use statistical methods such as an arithmetic encoder [52]. We also initially tried an arithmetic encoder to achieve higher compression ratio. We found that, by using the arithmetic encoder, we can improve a compression ratio by a factor of two, but the decompression performs about three times slower than using our dictionary-based method. Therefore, the proposed dictionary-based method improves the runtime performance by a factor of two for the tested benchmark applications over the

arithmetic encoder.

8 Conclusion and Future Work

We have presented a novel compression and runtime BVH decompression framework that transparently supports random access on the compressed BVHs. Our compression method preserves the original layout of a BVH and sequentially compresses BVs of a BVH. In order to support random access on the compressed BVHs, we decompose an input BVH into a set of clusters. Each cluster contains consecutive BV nodes and serves as an access point at runtime. We propose a general BVH access API to transparently support random access on our RACBVH representation. Our decompression framework selectively fetches, decompresses, and stores data in our in-core BVH representation. We have demonstrated the benefits of our methods on two applications having different characteristics. We achieved up to a 12:1 compression ratio and up to a 4:1 runtime performance improvement in the tested benchmarks.

There are many interesting avenues for future work. Our current method achieved two times performance improvement by supporting parallel random access on our compressed representations with four CPU-cores for ray tracing. Although this 50% parallel efficiency is higher than that achieved by accessing the original data, we would like to achieve a higher parallel efficiency by designing more compact in-core representations and aggressive compression methods. Also, we would like to apply our method to highly parallel architectures such as GPUs and Larrabee architecture [60]. Also, some of interactive ray tracers employ levels-of-detail (LOD) hierarchies. These LOD-based ray tracers can have very high space requirements due to the LOD hierarchy. We would like to apply our method to reduce the memory requirements of LOD hierarchies and design an interactive LOD-based ray tracer. Finally, our current method preserves the mesh layouts, causing our method to store meshes separately from BVHs. It may be possible to achieve a higher compression ratio and runtime performance by coupling meshes and BVHs, without preserving the original triangle layouts. It may be interesting to design such a method, while maintaining coherent mesh layouts within BVHs.

Acknowledgment

We would like to thank anonymous reviewers for their constructive feedbacks. We also thank Paul Merrell and members of KAIST SGLab. for their English review and helpful feedbacks. The St. Matthew and Lucy model are courtesy of Stanford University. The isosurface model is courtesy of the LLNL. The power plant model and CAD turbine models are courtesy of an anonymous donor and of Kitware respectively. This project was supported in part by MKE/MCST/IITA [2008-F-033-02,2009-S-001-01], MCST/KEIT [2006-S-045-1], MKE/IITA u-Learning, MKE digital mask control, MCST/KOCCA-CTR&DP-2009, KRF-2008-313-D00922, and MSRA E-heritage.

References

[1] J. L. Hennessy, D. A. Patterson, and D. Goldberg, *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, 2007.
[2] S. M. Rubin and T. Whitted, "A 3-dimensional representation for fast rendering of complex scenes," *Computer Graphics*, vol. 14, no. 3, pp. 110–116, 1980.
[3] M. Lin and D. Manocha, "Collision and proximity queries," *Handbook of Discrete and Computational Geometry*, 2003.
[4] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino, "Collision detection for deformable objects," *Computer Graphics Forum*, vol. 19, no. 1, pp. 61–81, 2005.
[5] P. M. Hubbard, "Interactive collision detection," *Proc. of IEEE Symposium on Research Frontiers in Virtual Reality*, pp. 24–31, 1993.
[6] G. van den Bergen, "Efficient collision detection of complex deformable models using AABB trees," *Journal of Graphics Tools*, vol. 2, no. 4, pp. 1–13, 1997.
[7] S. Gottschalk, M. Lin, and D. Manocha, "OBB-Tree: A hierarchical structure for rapid interference detection," *Proc. of ACM Siggraph*, pp. 171–180, 1996.
[8] J. Klosowski, M. Held, J. Mitchell, H. Sowizral, and K. Zikan, "Efficient collision detection using bounding volume hierarchies of k-dops," *IEEE Trans. on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 21–36, 1998.
[9] I. Wald, "Realtime Ray Tracing and Interactive Global Illumination," Ph.D. dissertation, Computer Graphics Group, Saarland University, 2004.
[10] P. Alliez and C. Gotsman, "Recent advances in compression of 3d meshes," *Advances in Multiresolution for Geometric Modelling*, pp. 3–26, 2004.
[11] C. Gotsman, S. Gumhold, and L. Kobbelt, "Simplification and compression of 3d meshes," in *Tutorials on Multiresolution in Geometric Modelling*. Springer, 2002, pp. 319–361.
[12] C. Touma and C. Gotsman, "Triangle mesh compression," in *Graphics Interface*, 1998, pp. 26–34.
[13] O. Devillers and P.-M. Gandoin, "Geometric compression for interactive transmission," in *IEEE Visualization*, 2000, pp. 319–326.
[14] J. Peng and C.-C. J. Kuo, "Geometry-guided progressive lossless 3d mesh coding with octree (ot) decomposition," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 609–616, 2005.
[15] P. Alliez and M. Desbrun, "Valence-driven connectivity encoding for 3D meshes," in *Eurographics*, 2001, pp. 480–489.
[16] F. Kälberer, K. Polthier, U. Reitebuch, and M. Wardetzky, "Freelence - coding with free valences," *Comput. Graph. Forum*, vol. 24, no. 3, pp. 469–478, 2005.
[17] M. Isenburg and J. Snoeyink, "Face Fixer: Compressing polygon meshes with properties," in *SIGGRAPH*, 2000, pp. 263–270.
[18] S. Gumhold and W. Strasser, "Real time compression of triangle mesh connectivity," in *SIGGRAPH*, 1998, pp. 133–140.
[19] J. Rossignac, "Edgebreaker: Connectivity compression for triangle meshes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, no. 1, pp. 47–61, 1999.
[20] H. Lee, P. Alliez, and M. Desbrun, "Angle-analyzer: A triangle-quad mesh codec," *Computer Graphics Forum*, vol. 21, pp. 383–392, 2002.
[21] S. Choe, J. Kim, H. Lee, S. Lee, and H.-P. Seidel, "Mesh compression with random accessibility," in *Israel-Korea Bi-National conf.*, 2004, pp. 81–86.
[22] S. Choe, J. Kim, H. Lee, and S. Lee, "Random accessible mesh compression using mesh chartification," *IEEE Trans. on Visualization and Computer Graphics*, vol. 15, no. 1, pp. 160–173, 2009.
[23] S.-E. Yoon and P. Lindstrom, "Random-accessible compressed triangle meshes," *IEEE Trans. on Visualization and Computer Graphics (Proc. Visualization)*, vol. 13, no. 6, pp. 1536–1543, 2007.
[24] E. Gobbetti, F. Marton, P. Cignoni, M. Di Benedetto, and F. Ganovelli, "C-bdam - compressed batched dynamic adaptive meshes for terrain rendering," *Computer Graphics Forum*, vol. 25, no. 3, pp. 333–342, 2006.
[25] J. Kim, S. Choe, and S. Lee, "Multiresolution random accessible mesh compression," *Eurographics*, vol. 25, no. 3, pp. 323–332, 2006.
[26] D. L. Gall, "Mpeg: a video compression standard for multimedia applications," *Communications of the ACM*, vol. 34, pp. 46–58, 1991.
[27] I. Ihm and S. Park, "Wavelet-based 3d compression scheme for interactive visualization of very large volume data," *Computer Graphics Forum*, vol. 18, no. 1, pp. 3–15, 1999.

[28] F. Rodler, "Wavelet based 3D compression with fast random access for very large volume data," in *Pacific Graphics*, 1999, pp. 108–117.

[29] S. Lefebvre and H. Hoppe, "Perfect spatial hashing," in *SIGGRAPH*, 2006, pp. 579–588.

[30] J. Katajainen and E. Makinen, "Tree compression and optimization with applications," *International Journal of Foundations of Computer Science*, vol. 1, no. 4, pp. 425–447, 1990.

[31] S. Zaks, "Lexicographic generation of ordered trees," *Theoretical computer science*, pp. 63–82, 1980.

[32] D. Zerling, "Generating binary trees using rotations," *Journal of ACM*, vol. 32, no. 3, pp. 694–701, 1985.

[33] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, 1991.

[34] D. Cline, K. Steele, and P. K. Egbert, "Lightweight bounding volumes for ray tracing," *Journal of Graphics Tools*, vol. 11, no. 4, pp. 61–71, 2006.

[35] J. Mahovsky, "Ray tracing with reduced-precision bounding volume hierarchies," PhD Thesis, University of Calgary, 2005.

[36] P. Terdiman, "Opcode: Optimized collision detection," 2003. [Online]. Available: http://www.codercorner.com/Opcode.htm

[37] S. Rusinkiewicz and M. Levoy, "Qsplat: A multiresolution point rendering system for large meshes," *SIGGRAPH*, pp. 343–352, 2000.

[38] E. Hubo, T. Mertens, T. Haber, and P. Bekaert, "The quantized kd-tree: Efficient ray tracing of compressed point clouds," in *IEEE Symp. on Interactive Ray Tracing*, 2006, pp. 105–113.

[39] D. Salomon, *Data Compression*. Springer, 2007.

[40] C. Lauterbach, S.-E. Yoon, and D. Manocha, "Ray-Strips: A Compact Mesh Representation for Interactive Ray Tracing," in *IEEE/EG Symposium on Interactive Ray Tracing*, 2007, pp. 19–26.

[41] C. Lauterbach, S.-E. Yoon, M. Tang, and D. Manocha, "ReduceM: Interactive and memory efficient ray tracing of large models," *Computer Graphics Forum (Proc. of EG Symp. on Rendering)*, vol. 27, no. 4, pp. 1313–1321, 2008.

[42] S. Lefebvre and H. Hoppe, "Compressed random-access trees for spatially coherent data," in *Eurographics Symposium on Rendering*, 2007, pp. 339–349.

[43] I. Wald, "On fast Construction of SAH based Bounding Volume Hierarchies," in *EG/IEEE Symposium on Interactive Ray Tracing*, 2007, pp. 33–40.

[44] I. Wald, S. Boulos, and P. Shirley, "Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies," *ACM Transactions on Graphics*, vol. 26, no. 1, p. 6, 2007.

[45] L. Arge, G. Brodal, and R. Fagerberg, "Cache oblivious data structures," *Handbook on Data Structures and Applications*, 2004.

[46] M. Isenburg and P. Lindstrom, "Streaming meshes," in *IEEE Visualization*, 2005, pp. 231–238.

[47] P. van Emde Boas, "Preserving order in a forest in less than logarithmic time and linear space," *Inf. Process. Lett.*, vol. 6, pp. 80–82, 1977.

[48] S.-E. Yoon and D. Manocha, "Cache-efficient layouts of bounding volume hierarchies," *Computer Graphics Forum (Eurographics)*, vol. 25, no. 3, pp. 507–516, 2006.

[49] C. Silva, Y.-J. Chiang, W. Correa, J. El-Sana, and P. Lindstrom, "Out-of-core algorithms for scientific visualization and computer graphics," in *IEEE Visualization Course Notes*, 2002.

[50] M. Isenburg and S. Gumhold, "Out-of-core compression for gigantic polygon meshes," in *SIGGRAPH*, 2003, pp. 935–942.

[51] S.-E. Yoon, D. Manocha, P. Lindstrom, and V. Pascucci, "Openccl," 2005. [Online]. Available: http://gamma.cs.unc.edu/COL/OpenCCL

[52] A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic coding revisited," *ACM Transactions on Information Systems*, vol. 16, no. 3, pp. 256–294, 1998.

[53] J. Klosowski, M. Held, J. S. B. Mitchell, K. Zikan, and H. Sowizral, "Efficient collision detection using bounding volume hierarchies of *k*-DOPs," *IEEE Trans. Visualizat. Comput. Graph.*, vol. 4, no. 1, pp. 21–36, 1998.

[54] A. Reshetov, A. Soupikov, and J. Hurley, "Multi-level ray tracing algorithm," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 1176–1185, 2005.

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

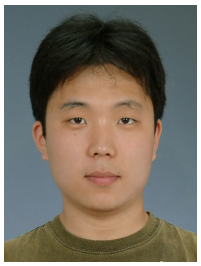
[56] R. L. Cook, T. Porter, and L. Carpenter, "Distributed ray tracing," in *SIGGRAPH*, 1984, pp. 137–145.

[57] B. Mirtich and J. Canny, "Impulse-based simulation of rigid bodies," in *Symposium on Interactive 3D Graphics*, 1995, pp. 181–188.

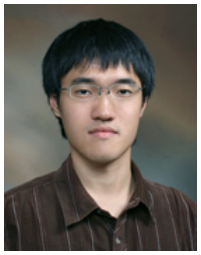
[58] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. Jon Wiley and Sons, 2003.

[59] J. loup Gailly and M. Adler, "zlib," 2005. [Online]. Available: http://www.zlib.net

[60] L. Seiler, D. Carman, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A many-core x86 architecture for visual computing," *ACM Transactions on Graphics*, vol. 27, no. 3, 2008.



Tae-Joon Kim is currently a Ph. D. student at KAIST. He received his B.S. degree in computer science from KAIST in 2007. His research interests include visualization, interactive rendering, and data compression.



Bochang Moon is currently a M.S. student at KAIST. He received his B.S. degree in computer science from Chung-Ang University in 2008. His research interests include visualization, interactive rendering, and global illumination.

Duksu Kim is currently a M.S. student at KAIST. He received his B.S. degree in information & communication engineering from Sung Kyun Kwan University in 2008. His research interests include collision detection, motion planning, and parallel computing.

Sung-Eui Yoon is currently an assistant professor at KAIST (Korea Advanced Institute of Science and Technology). He received the B.S. and M.S. degrees in computer science from Seoul National University in 1999 and 2001 respectively. He received his Ph.D. degree in computer science from the University of North Carolina at Chapel Hill in 2005. He was a postdoctoral scholar at Lawrence Livermore National Laboratory. His research interests include visualization, interactive rendering, geometric problems, and cache-coherent algorithms and layouts. He is particularly interested in designing scalable algorithms that can handle massive models in commodity hardware. He wrote a monograph on real-time massive model rendering with other three co-authors. He is a member of IEEE, ACM, and Eurographics.