FlashBack: Immersive Virtual Reality on Mobile Devices via Rendering Memoization

Kevin Boos
Rice University
kevinaboos@rice.edu

David Chu
Microsoft Research
davidchu@microsoft.com

Eduardo Cuervo
Microsoft Research
cuervo@microsoft.com

ABSTRACT

Virtual reality head-mounted displays (VR HMDs) are attracting users with the promise of full sensory immersion in virtual environments. Creating the illusion of immersion for a near-eye display results in very heavy rendering workloads: low latency, high framerate, and high visual quality are all needed. Tethered VR setups in which the HMD is bound to a powerful gaming desktop limit mobility and exploration, and are difficult to deploy widely. Products such as Google Cardboard and Samsung Gear VR purport to offer any user a mobile VR experience, but their GPUs are too power-constrained to produce an acceptable framerate and latency, even for scenes of modest visual quality.

We present FLASHBACK, an unorthodox design point for HMD VR that eschews all real-time scene rendering. Instead, FLASHBACK aggressively precomputes and caches all possible images that a VR user might encounter. FLASHBACK memoizes costly rendering effort in an offline step to build a cache full of panoramic images. During runtime, FLASHBACK constructs and maintains a hierarchical storage cache index to quickly lookup images that the user should be seeing. On a cache miss, FLASHBACK uses fast approximations of the correct image while concurrently fetching more closely-matching entries from its cache for future requests. Moreover, FLASHBACK not only works for static scenes, but also for dynamic scenes with moving and animated objects.

We evaluate a prototype implementation of FLASHBACK and report up to a 8× improvement in framerate, 97× reduction in energy consumption per frame, and 15× latency reduction compared to a locally-rendered mobile VR setup. In some cases, FLASHBACK even delivers better framerates and responsiveness than a tethered HMD configuration on graphically complex scenes.

1. INTRODUCTION

Driven by recent advances in the mobile computing hardware ecosystem, wearable Virtual Reality (VR) is experiencing a boom in popularity, with many offerings becoming available. Wearable VR head-mounted displays (HMDs) fall into two device classes: *(i) Tethered HMDs:* HMDs tethered to powerful, expensive gaming desktops, such as the Oculus Rift, HTC Vive, and Sony Playstation VR; *(ii) Mobile-rendered HMDs:* Self-contained, untethered HMDs that run on mobile phones slotted into head mounts, e.g., Google Cardboard and Samsung Gear VR.

However, both device classes present significant drawbacks. Tethered HMDs are capable of rendering rich graphical scenes at high framerates and visual quality, but require significant GPU and compute resources in the form of a dedicated gaming desktop or console co-located with the user. Tethered HMDs obviously limit mobility and come with a high barrier to entry, but also suffer the risk of tethered cords wrapping around a user's neck.

Mobile-rendered HMDs are widely available but suffer from low graphical quality, poor battery life, and uncomfortable thermal radiation, all of which break the illusion of immersion. Mobile GPU rendering can consume up to 20W of peak power [20], making thermal output a safety concern for near-eye devices without active cooling. Limiting mobile GPU power (and thus, performance) is highly undesirable because *(i)* mobile GPUs are already over an order of magnitude slower than desktop GPUs, and *(ii)* a near-eye display exacerbates any performance degradations, often causing motion discomfort or simulator sickness.

In addition, we believe that affordability is vital to widespread VR adoption. Tethered HMDs are clearly cost-prohibitive, but even mobile-rendered HMDs require high-end phones with high-end GPUs. Providing immersive VR experiences on widely available, affordable devices will enable exciting new use cases: virtual field trips for low-income or remote classrooms, enhanced training simulations, medical education and examination, therapeutic rehabilitation, and many more beyond VR gaming [6].

The FLASHBACK Design

In this paper, we present FLASHBACK, a system that overcomes the limitations of both Tethered and Mobile-rendered HMDs to offer a full-quality VR experience on weak mobile devices. FLASHBACK does so by serving *all* of a VR application's high data rate rendering requests from a local cache of pre-rendered HD frames, effectively memoizing prior rendering efforts. We are agnostic as to what machine generates the cache — it could be a dedicated cloud rendering server, a nearby desktop, or the HMD device itself (given plenty of time) — as long as the cached contents can be downloaded to the HMD device before run time.

Pre-caching avoids the struggle of real-time rendering on a weak mobile GPU while leveraging a prevailing trend among mobile devices: *storage is low-power, increasingly abundant, cheap, and often underutilized*, while graphical processing remains restricted due to thermal and energy constraints. In fact, we show that storage is sufficient to fully cache entire VR scenes.

MobiSys'16, June 25-30, 2016, Singapore, Singapore
© 2016 ACM. ISBN 978-1-4503-4269-8/16/06. . . $15.00
DOI: http://dx.doi.org/10.1145/2906388.2906418

Moreover, FLASHBACK fundamentally changes how VR applications can be deployed and executed on mobile devices. Instead of running the application binary itself, one simply downloads the application's pre-rendered results (or generates them locally ahead of time) for future use during playback, similar to downloading a movie. However, unlike a movie, the VR experience is *highly* non-linear and interactive.

FLASHBACK builds a three-tier frame cache across GPU video memory (VRAM), system RAM, and secondary storage to store the set of frames needed for a given VR application. The cache is indexed by the player's current *pose*, its 3D position in the environment. As the player moves around, FLASHBACK retrieves and displays a new frame from the cache that matches the updated pose, using a nearest-neighbor algorithm to quickly search the 3D space. Based on R-trees, our index is optimized to quickly return results in GPU memory for immediate display while concurrently fetching better cache entries from deeper in the storage hierarchy for future requests. On a cache miss, FLASHBACK uses cheap and fast approximations of the correct image based on well-established mesh warping techniques from the computer graphics community [24]. We further introduce cache compression techniques to not only fit more cache entries in storage, but also to increase system throughput. Section 4 explains the layout, usage, creation, and compression of the frame cache in greater detail.

In addition to handling a static background scene, FLASHBACK even supports dynamically-moving, animated objects (e.g., a person walking, or *n* cars driving) using a per-object cache data structure. Dynamic object caches are indexed by the object's animation stage, orientation, and relative distance from the player pose for a given time- or movement-based trigger. Unlike the static scene cache, a dynamic object cache stores frames that contain a view of the dynamic object only, allowing FLASHBACK to combine the static frame with multiple dynamic frames using pixel depth metadata embedded in each frame. With support for both static scenes and dynamic objects, FLASHBACK can handle many types of VR applications. Section 5 provides a deeper exploration of dynamic animated objects.

We develop a prototype implementation of FLASHBACK on Windows 10 that supports rendering memoization of VR applications created with Unity, the most popular commercial game and virtual reality creation tool. Our implementation is in three parts: a Unity-side instrumentation suite that automates offline cache generation, a `CacheManager` library that controls local cache contents on the HMD, and a runtime based on DirectX 11 that issues cache queries, composites cache entries into final scenes, and displays rendered content onto the HMD. Our implementation does not require modifying the Unity VR application or the HMD's VR drivers, as described in Section 6.

Finally, we investigate the performance limits of FLASHBACK with a thorough evaluation of our prototype on an Oculus Rift VR headset powered by a weak HP Pavilion Mini device. FLASHBACK achieves up to a 15× reduction in end-to-end latency, an 8× increase in overall framerate, and a 97× reduction in per-frame energy consumption compared with a Mobile-rendered configuration running a complex, fully-fledged VR environment. The graphical quality, framerate, and latency of FLASHBACK are even on par with — and sometimes better than — that of a strong gaming desktop. We also show that FLASHBACK's cache can scale to large virtual environments and can handle a reasonable number of concurrently visible dynamic objects. Therefore, FLASHBACK is well-positioned to bring immersive VR experiences to mobile devices through its novel rendering memoization techniques.

As VR is a visceral experience better seen than described, we provide a video demonstration of FLASHBACK at [26].

2. BACKGROUND

Basic Operation of a VR System: A modern HMD, like the Oculus Rift, has a variety of internal sensors, e.g., IMUs, that track the player's *pose*, comprised of 3D *position* and 3D *orientation*. Some systems also have external sensors or cameras that track the position of the HMD on the user's face with respect to the surrounding physical room. The display of the HMD is often the same class as those used on smartphones and tablets. HMDs use the tracked position to render the corresponding virtual environment to the display.

Virtual Environment Creation Tools: The virtual environment can be rendered from any computer-generated scene. Commercial virtual environment creation tools are often the same as game creation IDEs, such as Unity and Unreal. These IDEs provide a convenient WYSIWYG way to rapidly construct and script scenes based on pre-existing game objects. We leverage two important properties of these tools. First, they clearly delineate *static objects* from *dynamic objects*. Static objects in the virtual environment do not change. Examples include buildings, terrain, landscape and other immutable objects. A *static scene* consists of all static objects rendered together. Dynamic object examples include vehicles, animals, people, and anything with motion animations. Second, the camera that generates the rendered result is conceptually abstracted from the scene. As a result, it is straightforward to replace a scene's camera with a custom camera, which we do to generate cache entries.

VR as a mobile workload: VR HMD systems place heavy rendering and power demands on computing systems. Modern VR systems target:

- *low latency:* total end-to-end (motion-to-photon) latency of under 25ms, half that of previous VR systems;
- *high framerate:* throughput of at least 60 frames per second (FPS) to ensure smooth playback;
- *scene complexity:* visually rich, photo-realistic scenes.

These requirements are among the most demanding for consumer mobile applications. In the temporal dimension, the latency and framerate requirements derive from the fact that we are physiologically very sensitive to lag in near-eye displays because the human visual and vestibular sensory systems are tightly coupled. Even minor motion-to-photon latency can induce oscilloscopia (the sensation that your view and vestibular signals are mismatched), and eventually motion sickness or simulator sickness [7, 22, 4]. While classic studies found tolerance thresholds of 50ms (which coincided with measurement resolution) [3], more recent anecdotal evidence suggests that 10-20ms is a better target, depending upon the scene and user [25].

In the spatial domain, *scene complexity* refers to substantive detail in a graphical scene, such as rich geometry and texture detail. A near-eye display intensifies the importance of scene complexity because the HMD's pixels are mere centimeters from the eye and magnified on the retina; thus, graphical detail (or lack thereof) becomes immediately more noticeable. Delivering photo-realistic scenes requires substantial GPU processing capabilities.

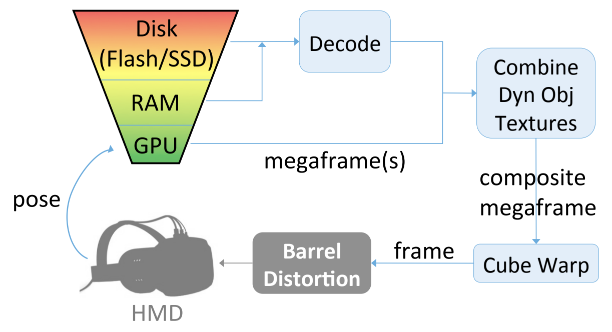


Figure 1: FlashBack runtime operation. FlashBack uses the HMD pose to query the cache and decode the retrieved megaframe if necessary, then it combines that static megaframe with any dynamic object megaframes, and finally warps the megaframe into a single final frame for the HMD.

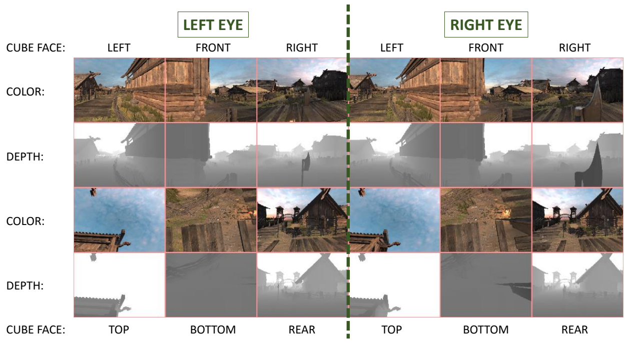


Figure 2: Each cache entry contains a megaframe. The twenty-four faces of the megaframe represent the left and right eye cube maps for RGB color and depth.

The core challenge facing an untethered mobile HMD system such as FlashBack is to provide all of the above properties — low latency, high framerate and graphical quality — simultaneously at low power. Unfortunately, in Mobile-endered (and even Tethered) HMDs, latency and high framerate are at odds with resolution and scene complexity: striving for higher quality scenes impinges upon latency and framerate, and vice versa.

3. SYSTEM OVERVIEW

Figure 1 depicts the high-level operation of FlashBack, from sampled input to displayed output. First, the current player *pose* is read in from the HMD driver, comprising *position* and view *orientation*. The position is the location of the player in 3D world space; the view orientation is a rotation vector that represents where the player is looking.

FlashBack then finds and reads multiple cache entries that are needed for the user's view. One of these cache entries corresponds to the static scene and the other entries correspond to the dynamic objects in the scene. The cache lookup encompasses GPU memory, system memory, and non-volatile secondary storage, with varying levels of access speed. We optimize this lookup with cache indexing (Section 4.4) and cache compression (Section 4.3). When required, entries are pulled from higher to lower levels of the cache hierarchy, evicting older entries. The matched cache entries are then composited into a final view.

Upon a cache miss, instead of rendering the correct view in real time, we synthesize an approximation of the correct view from available cache entries with a computer graphics technique known as mesh warping (Section 4.5). Warping is significantly lighter-weight than rendering. Most importantly, unlike rendering, warping speed is not dependent on scene complexity; it is only a fixed function of the screen resolution and runs efficiently even on mobile GPUs. As a result, the scene can have arbitrarily complex visual detail and effects, yet warping speed remains constant.

As a final step, the HMD device driver performs lens-offsetting barrel distortion and displays the final frame to the screen. The components of this entire process are the main contributors to the system's end-to-end motion-to-photon latency, so we strive to make them as efficient and performant as possible.

4. CACHE ORGANIZATION

In this section, we describe the overall design of FlashBack's rendering memoization mechanism, its cache structure and behavior, and optimizations for scalability and performance of static scenes.

4.1 A Single Cache Entry: The Megaframe

Each entry of the cache consists of a high resolution *megaframe* as shown in Figure 2. A megaframe is defined with respect to a pose $\mathbf{p} = ((x, y, z), (\theta, \phi, \psi))$. The parameters (x, y, z) represent the position in 3D world coordinates. The parameters (θ, ϕ, ψ) represent the orientation (sometimes referred to as rotation) as a Euler angle comprising yaw, pitch, and roll, respectively. With appropriate warping (Section 4.5), the megaframe allows us to reconstruct nearby views that are translated or rotated with respect to the megaframe's pose.

Internally, a megaframe is composed of four *cube maps*. A cube map is a classic computer graphics 360° representation of an environment [14]. The cube map draws a panoramic image on the six sides of a cube, with the centerpoint of the cube being the current pose. The four cube maps in a single megaframe include:

- Left eye color (RGB) cube map,
- Left eye depth cube map,
- Right eye color (RGB) cube map, and
- Right eye depth cube map.

The left and right eye cube maps exist separately in order to generate a proper stereo view. Their positions are each offset from the megaframe's pose by half the inter-pupillary distance (IPD), which is a user-specific anatomical property that represents the distance between human eyes. The depth cube maps are not necessary for representing the RGB pixel content of the scene, but are useful during the warping step. All four cube maps in every megaframe are stored consistently at a fixed, canonical orientation looking straight ahead, i.e., $(\theta, \phi, \psi) = (0, 0, 0)$. With four cube maps and six faces per cube, the megaframe consists of 24 faces, as illustrated in the megaframe layout of Figure 2.

4.2 Cache Layout and Hierarchy

Figure 3 provides a visualization of how the cache is laid out in logical 3D space. The megaframes conceptually occupy the 3D point matching the pose at which they were rendered; as a

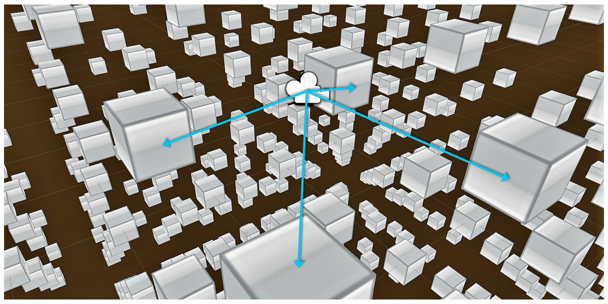


Figure 3: Logical layout of megaframes. Each cube represents a megaframe's cache index visualized in 3D space. Arrows depict the distance from the closest few megaframes to the current pose query, represented by the camera icon.

player (camera icon) moves throughout the environment, it becomes closer to certain cubes and further from others. Distance is defined with respect to the position difference in Euclidean space. It is not necessary to consider orientation differences since megaframe cube maps are panoramic; in fact, cube maps inherently contain all possible orientations for a given position.

In terms of physical layout in memory or on storage, FLASHBACK builds a three-tier cache of megaframes, depicted in Figure 1 as an inverted triangle consisting of GPU VRAM as L1, system RAM as L2, and persistent secondary storage, like SSD or Flash, as L3. Although the size of each tier in Figure 1 is not to scale, GPU VRAM is the smallest, followed by a larger-sized RAM, and finally a massive secondary storage unit. Current mobile SoCs have GPU VRAMs statically allocated from system memory, typically a few hundred MBs on integrated chips. System RAM is usually 0.5–2GB (excluding GPU VRAM carve out). Secondary storage sizes of flash can be up to several hundred GBs. SSDs, a composition of multiple flash chips, can be thousands of GBs. While SSDs are not common on mobile devices today, they are worth consideration as they provide a point of extrapolation for future device storage trends. As such, a moderate number of megaframes are on the GPU VRAM, while all other megaframes are relegated to the other two layers.

We initially considered how the physical storage layout of cached frames would affect performance, believing that retrieving a cached frame from disk could incur a substantial and unpredictable latency penalty due to random reads with poor locality. However, as demonstrated in §7, decoder latency (explained below) dominates storage read latency by 2-3 orders of magnitude. Therefore, we find it unnecessary to optimize the cache's on-disk layout.

4.3 Cache Compression

Cache compression provides for better performance and spatial efficiency. Were we to store megaframes in a raw, uncompressed format, we would rapidly saturate the data transfer bandwidth between stages [9], as megaframes must be passed between GPU VRAM and system RAM. Saturation leads to low framerates and high latency. Note that even though GPU and system memory share the same physical memory banks in mobile SoCs, data transfer between the two still entails data copy because of format incompatibility and pointer swizzling. Therefore, we elect to store megaframes in a compressed format (equivalent to a single-frame H.264 video) when in system memory and stable storage. We only decompress frames in GPU memory when they are most likely to be displayed to the user. For efficiency, we leverage the dedicated hardware H.264 decoder available in all modern devices (typically used for video playback).

Another benefit of storing encoded frames on stable storage is that each cache entry is smaller in size. As an example, a decoded 4k texture consumes over 8MB of memory, but encoding that texture reduces it to under 100KB, allowing FLASHBACK to maintain vastly larger caches.

Even with frame compression at the L2 and L3 layer, the performance gap between L1 access and L2 or L3 access is large, as demonstrated in Section 7. This is because decoding frames still takes time, even with a dedicated hardware-accelerated decoder. On balance, trading data transfer time for decompression time and an increase in the maximum number of cache entries is an important part of the FLASHBACK design.

4.4 Cache Lookup and Indexing

FLASHBACK appoints a `CacheManager` to control the behavior of the cache and the flow or eviction of megaframes between different cache levels. The `CacheManager`'s primary function is to accept a request in the form of a `CacheKey` (`CK`) structure, containing player pose, and return a `CacheValue` (`CV`) structure containing a reference to a retrieved megaframe, decoding it if necessary. The arrows in Figure 3 show the Euclidean distance vectors used to locate the closest matching megaframe cube for a given requested `CK` pose.

We realize this querying semantic via a *nearest-neighbor search* using R-trees [17]. The R-tree algorithm constructs a set of minimally-overlapping bounding boxes that each contain subsets of points (in our case, the megaframe positions) in the 3D space, helping to rapidly eliminate large portions of the search space. When the correct box is located, the algorithm calculates the distance from each existing point to the target point (the desired pose's position) and selects the closest one. We choose R-trees because they support: *(i)* fast lookup; *(ii)* queries across storage hierarchies, better than other nearest-neighbor indexes like quad-trees and kd-trees, and *(iii)* good support for insertions and deletions.

We design our use of R-trees in such a way that whenever we receive a new pose request, we can always immediately return a megaframe result from the GPU cache for display. At the same time, if there is an even closer megaframe that exists in either L2 or L3 cache, it is fetched asynchronously to the GPU such that it is available to service future pose requests, taking advantage of temporal locality. As such, the notion of a cache miss refers to the requested megaframe not having an identical match in the GPU cache.

To support this goal, we use a dual R-tree data structure. Specifically, we maintain two distinct R-trees: a *GPU R-tree* and a *universal R-tree*. The GPU R-tree only indexes cache entries that are currently resident in the GPU cache, whereas the universal R-tree indexes all cache entries across all three storage levels. A pose request is issued to both the GPU R-tree and universal R-tree in parallel. The nearest neighbor megaframe in the GPU R-tree is returned immediately for display. The nearest neighbor megaframe in the universal R-tree is also looked up. If it is the same as the megaframe returned from the GPU R-tree, no further action is taken. If it differs, it is then transferred from secondary storage (if it was on L3, from RAM if L2) and then decoded to L1 asynchronously. When a new megaframe is decoded, it is inserted into the GPU R-tree and updated in the universal R-tree.

FLASHBACK's CV structure must be kept to a minimal size because there are potentially millions of instances, one for every cached megaframe. A CV holds a pointer to either a file location on persistent storage (L3), byte array in system memory (L2), or raw texture on GPU VRAM (L1), depending on which cache level it resides. In fact, a CV can exist in multiple cache levels simultaneously, offering redundancy if the CV must be evicted from VRAM or RAM cache to relieve memory pressure. Since our cache contents are read-only, we never need to write back cache entries into stable storage. Furthermore, cache eviction is simply a matter of removing a cache entry from the index, a fast operation for R-trees. We currently provide a flexible eviction mechanism and a simple LRU policy, but future policies could be more intelligent, e.g., evicting the furthest cache entry from the player's current position.

4.5 Cache Miss and Approximate Results

An embedded assumption in FLASHBACK is that every possible rendering request can be served by cached contents in one of the cache layers. Of course, even plentiful stable storage is finite. Therefore, in order to handle cache misses, we reuse nearby cached entries to approximate the desired result. This allows us to substantially increase FLASHBACK's effective cache hit rate.

However, naïvely substituting a view centered at pose $\mathbf{p}$ in lieu of a desired view at pose $\mathbf{p}'$ results in a poor experience with uncomfortable visual stuttering. Therefore, we apply a mesh warp to the megaframe at $\mathbf{p}$ in order to derive an appropriate view for $\mathbf{p}'$. Mesh warping is a classic technique from the family of computer graphics techniques known as Image-Based Rendering (IBR) [24]. We explain the mechanics, limitations, and advantages of mesh warp below.

Given an RGB cube map and matching depth cube map both at pose $\mathbf{p}$ (say, of the left eye), we can generate a novel view v' as if it had been taken from a new pose $\mathbf{p}'$. At a high level, each pixel of the original view is mapped to a 3D position (since $\mathbf{p}$ and the depth map are known), and then the 3D position is reprojected to a pixel in the new view (since $\mathbf{p}'$ is known). The final view v' resolution is proportional to the size of the megaframe. Assuming a typical HMD field of view (106° height, 94° width), a 4k megaframe (3840 × 2160) generates 720p final view frames (1280 × 720).

However, if translation is too great (i.e., the position of $\mathbf{p}$ and the position of $\mathbf{p}'$ are too far apart) then v' will suffer from *visual artifacts* such as disocclusions. Imagine looking at an open doorway and then stepping forward; from the original view, it is unclear what should appear in the disoccluded "holes" that are now visible. This suggests that we may desire additional cube maps to handle translations that are beyond a threshold, which is precisely what our additional megaframes provide. On the other hand, since our cube map covers a panoramic view, mesh warping is robust to arbitrary changes in rotation without introducing artifacts.

4.6 Populating the Cache

We now discuss how FLASHBACK actually generates the megaframes that will occupy the cache. These frames are generated offline, either on the mobile device itself (given enough time) or alternatively downloaded much like a video file from a desktop computer or powerful rendering server in the cloud. Deploying a dataset as large as the megaframe cache from a cloud server to the mobile device seems prohibitive at first, but is in actuality quite tractable due to the cache's extremely high compressability. The cache can be greatly compressed on the server due to adjacent megaframes having largely identical blocks, and then decompressed (decoded) on the mobile device in an ahead-of-time cache unpacking step.

Logically, FLASHBACK performs a 3D grid sweep across the virtual space constituting the static scene. At each grid point, FLASHBACK captures a panoramic stereo image of the world and writes this to a cube map. It does this again for depth, and then composites the corresponding megaframe. The megaframe is then encoded as an individual key frame (I-frame) using the H.264 codec. Finally, FLASHBACK writes the encoded megaframe to secondary storage with a unique identifier linking back to the pose from which it was generated. This fully-automated procedure repeats for every possible pose in the environment, which is potentially n^3 combinations due to the three dimensions of the pose's position value. The density of the grid, or *quantization*, impacts both the final cache size and the visual artifacts encountered during the warping approximation, as well as latency and framerate. We found that a virtual grid density between 0.02 and 0.05 virtual-world units (e.g., 2-5cm) offers a good trade-off between unnoticeable visual artifacts and cache size (§8).

Furthermore, we can aggressively cull the set of possible pose values based on the geometry and restricted movement paths of the environment. For example, for a virtual environment in which the player walks on the ground, we can limit the potential height values to a smaller range, e.g., five to seven feet above the ground. This technique significantly reduces the pose state space by eliminating impossible values, such as being underground or inside of a solid wall. Thus, while the worst case complexity of generating the megaframe cache is $O(n^3)$, the typical case is much less.

5. HANDLING DYNAMIC OBJECTS

In addition to caching the static environment's megaframes, FLASHBACK supports dynamic objects complete with freeform motion paths and animations. Dynamic object caching extends the base semantics of static caching with a more complex cache key structure and querying procedure.

5.1 Generating the Dynamic Object Cache

Rendering memoization for dynamic objects involves a procedure similar to that of static scenes. Offline, FLASHBACK iterates over the input space and renders megaframes. However, instead of only iterating over possible positions, dynamic objects have more dimensions: position, orientation, and animations. This results in a massive input space for each object, but fortunately it can be pruned along all three dimensions.

Capturing Dynamic Object Megaframes

We now describe the full procedure for populating a dynamic object's cache. As a preprocessing step, we extract and treat each dynamic object independently by placing it in an otherwise empty virtual world. This is important for megaframe composition free from side effects, described in Section 5.3 below.

Next, we iterate over all possible values along the position, orientation and animation dimensions in a set of nested loops. The

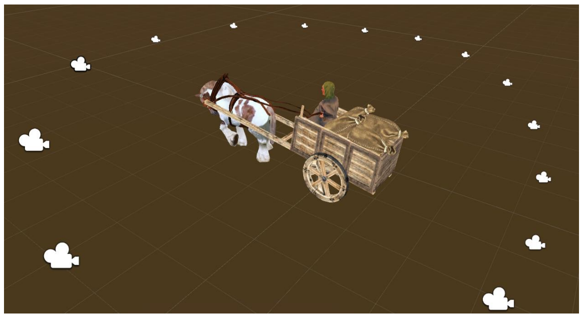


Figure 4: FLASHBACK caches dynamic objects by rendering a megaframe from all possible relative player positions and viewing angles, depicted here as camera icons. This repeats for all object animations and rotations.

outer loop is position. This position value consists of the same 3D point format but has a different semantic meaning from a static megaframe's position: it is the position of the player *relative* to that of the object, which is calculated by the Euclidean distance between the object position and the player camera position. We prune the position dimension in the same way as for static frames: the dynamic object is only visible to the player from a limited geographical area (e.g., distance). It is important to understand that the position value used for dynamic objects is *not* the physical location of the dynamic object itself in world space, but rather the position of the player relative to the dynamic object. This vastly reduces the position-dimension state space. For example, when the player stands 5 meters north (in a bird's-eye view) of the dynamic object, this configuration results in the same player view of that object no matter the absolute position of the two bodies in the virtual environment.

The next inner loop iterates over all possible values along the orientation dimension. Just like the player, a dynamic object can have its own view orientation, commonly referred to as the object's rotation. We prune the orientation dimension according to both the potential player viewpoints as well as the possible rotations of the object itself. For example, if the object rotates along the vertical y-axis only, a very common behavior for animated objects, we only need to iterate over angles in the y direction.

The final inner loop iterates over all possible *animation stages* in the object's animation sequence. There are up to as many stages as there are frames in the animation sequence; stages can be a downsampling of the number of frames in the animation. For example, the horse-drawn cart in Figure 4, which is a professionally produced dynamic object with animations, has a detailed "trotting" animation sequence, but we were able to represent it as a periodically repeating set of 36 stages with no loss in animation fidelity. If an object has multiple animation sequences (e.g., walking, running, or standing), the procedure is repeated for every sequence.

Capturing Dynamic Object Pose Traces

In order to know which megaframe should be used for a dynamic object at a given point in time, FLASHBACK records a *pose trace* of the dynamic object during offline playback of the original VR application. This trace defines the object's motion path and stage in the animation sequence for a given timestamp. Note

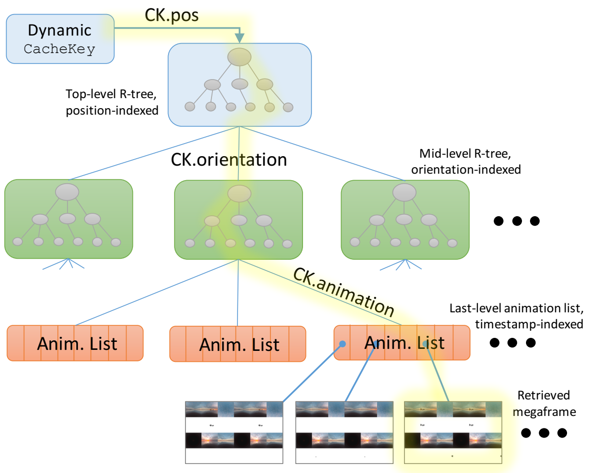

Dynamic CacheKey
CK.pos
Top-level R-tree, position-indexed
CK.orientation
Mid-level R-tree, orientation-indexed
CK.animation
Last-level animation list, timestamp-indexed
Anim. List
Anim. List
Anim. List
Retrieved megaframe

Figure 5: Nested R-Trees for Retrieval of Dynamic Objects. Dynamic objects are indexed by relative position (top-level R-tree), orientation (nested R-trees), and animation stages (simple lists).

that this means dynamic objects appear deterministically based on the timeline. For example, a horse always trots along the same path in the virtual environment, independent of where the user is positioned or looking. Future work includes extending the cache index to support different dynamic object behavior based on the player's actions and pose.

5.2 Dynamic Object Index and Lookup

The query execution for dynamic object caches is a two-step process. First, the object's pose trace is queried using the current timestamp to determine at what view orientation and animation stage that object should appear.

Once the dynamic object pose descriptor is acquired, step two is to query the actual megaframe cache using a *dynamic cache key* (dynamic `CK`), consisting of the pose descriptor from the trace in step one (orientation and animation) and the relative player position. The dynamic `CK` consists of 7 values: a 3D position vector, 3D orientation vector, and scalar animation stage.

While it is possible to construct a 7 dimension R-tree, it would suffer because a high-dimension data structure is ill-suited for use with spatial indexing algorithms. Instead, we construct a *nested R-tree* as shown in Figure 5 that reduces the R-tree to 3 dimensions at most. It consists of a top-level R-tree indexed by position (similar to the universal R-tree for static scenes). However, instead of its leaf nodes pointing to megaframes, each leaf node points to a second-level R-tree indexed by orientation; there are n such orientation R-trees, one for each node in the position R-tree. Finally, each leaf node of the orientation R-tree points to a simple list that maps a timestamp to an animation stage pointer. Finally, the animation stage pointer points to a megaframe that was captured with the corresponding 7-tuple.

The logic behind the nesting ordering of position, orientation and animation stage is that it prioritizes position distance. Suddenly seeing a large translation jump in a dynamic object will throw off the player's inner balance more so than seeing that object at an unexpected rotation angle or a strange animation frame.

To execute a query, the `CacheManager` first queries the top-level (position-indexed) R-tree using the relative position value to

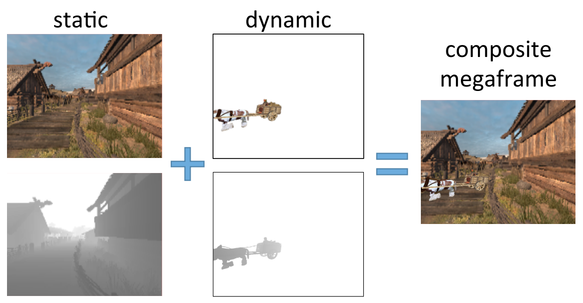


Figure 6: Compositing Dynamic Objects With Static Scenes. Croppings of the megaframes (a single cube face) for one static scene and one dynamic object are shown.

obtain a reference to the mid-level R-tree of orientation indices. It then queries that mid-level R-tree using the dynamic object's current orientation to obtain a reference to the lowest level list of animation frame sequences, which is finally indexed using the animation stage timestamp to obtain the actual megaframe that best matches the dynamic `CK`'s values.

5.3 Compositing Dynamic Objects

Once a megaframe has been retrieved for every visible dynamic object, all of their megaframes along with the static scene megaframe are overlaid into a single composite megaframe. Recall that each megaframe includes not only the cube faces representing the actual scene pixels, but also the depth values for each pixel (see Figure 2). We utilize these depth values in a pixel shader to determine which objects should occlude other objects and components of the static scene, and vice versa. Therefore, FLASHBACK can support a complex layering of both dynamic and static objects, e.g., placing a sprite in front of one building but behind another. An illustrative example is shown in Figure 6. After composition is complete, the composite megaframe is handed off to the cube map warp routine to construct the final view.

5.4 Look-ahead Decoding

When we have idle decode cycles (e.g., when many requests are hitting the GPU cache), it is worthwhile from an energy and performance perspective to speculate on which megaframes will be requested in the near future. That way, we can preemptively decode the chosen megaframes and have them ready on GPU VRAM by the time they are needed. We employ a relatively straightforward speculation mechanism that looks n frame periods into the future (e.g., 5 frames from now) by reading n future values of the dynamic object's pose trace. FLASHBACK then instructs the `CacheManager` to lookup those n future megaframes and decode as many as time permits.

Though this is particularly well-suited for dynamic objects due to their predictable motion pattern known ahead of time, we can also apply it to the static scene by speculating on the player's movement. This turns out to be less successful than for dynamic objects (§7) because it relies on accurately predicting the user's future pose, which has much more uncertainty than deterministic dynamic object poses [21].

5.5 Limitations of Dynamic Objects

As FLASHBACK composites the pixel representation of dynamic objects, it is more limited than general rendering.

Lighting Models: Our approach cannot support certain lighting models and special effects for dynamic objects: these include scene-dependent lighting and reflections. For example, if the appearance of a dynamic object depends on where it is relative to the scene's light sources, our caching mechanism will fail to capture the object's lighting variations. These limitations stem from the fact that we construct the cache entries for the dynamic object in isolation from the static scene. It is conceivable that the cache can also be augmented to include indexing by relative light source positions and orientations, much like it is already indexed by relative player pose. We leave this as future work.

Scalability: FLASHBACK's technique for handling dynamic objects has unique scalability properties; we quantify and explain its behavior in Section 8.2.

6. IMPLEMENTATION

Our prototype implementation of FLASHBACK runs on Windows 10 using the Oculus Rift DK2 HMD. It is powered by a weak HP Mini computer equivalent in compute and GPU power to a mid-range smartphone (§7). Our implementation is in three parts: the Unity cache generator that automates the rendering memoization process, the `CacheManager` on the HMD, and the lightweight playback client on the HMD.

Unity-side Cache Generation: We implement Unity-side cache generation by adding a special array of rendering cameras that automatically generate megaframes. A series of scripts coordinate the camera behavior with the automated pose enumeration (§5.1), which can be either manually bounded or automatically inferred based on the collision boxes and environment geometry of the Unity application. Every megaframe is encoded with an external ffmpeg toolchain and saved to cache storage. In total, we package 3900 lines of C# code into a Unity *prefab* that simplifies the incorporation of our caching scripts down to a simple drag-and-drop import operation. Thus, any Unity game or VR application can be quickly and easily transformed to work with FLASHBACK.

CacheManager: We implement the `CacheManager`, including all querying, organization, and handling of cached megaframes, with approximately 1200 lines of C++ code. In addition to the functions described in Sections 4 and 5, the `CacheManager` provides raw megaframes to the playback program for display, meaning that it must handle decoding. The `CacheManager` is also responsible for parsing the cache contents from storage. Because FLASHBACK's caches can scale to very large proportions, parsing them must also perform well at scale. Iterating through every cached file is prohibitively slow, so FLASHBACK creates a special index file for each cache instance once the cache is fully generated, enabling the playback program to bypass the parsing process altogether. Instead, it memory maps or reads each index file directly into a pre-allocated R-tree instance, using an *initial packing algorithm* to create the whole R-tree in one batch, offering better query performance as a side benefit.

Lightweight Playback Program: The cache population procedure effectively flattens a VR application's complex behavior into a collection of data structures on storage. As such, the program needed to "play" the application on the mobile device is relatively

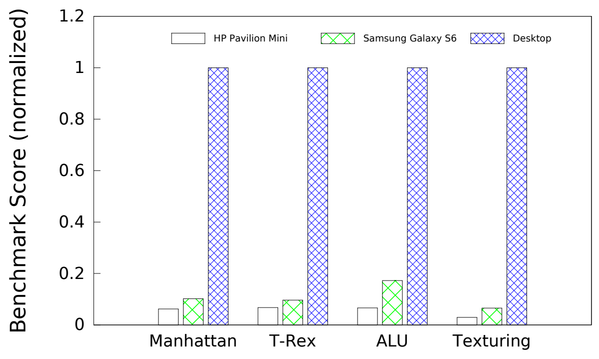


Figure 7: Four GFXBench benchmarks across three devices demonstrate that the HP Pavilion Mini is less capable than a Galaxy S6, and that both devices are an order of magnitude weaker than a VR-capable desktop [1].

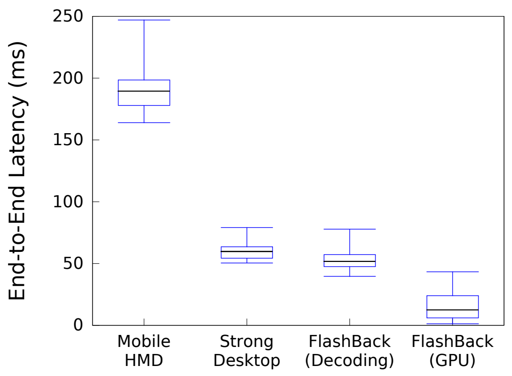


Figure 8: FLASHBACK achieves an end-to-end latency up to 15X shorter than a weak mobile device rendering locally, with worst-case latency no higher than that of a gaming desktop.

simple. We develop a reference implementation of the playback program atop a remote desktop client framework with skeleton code for decoding and displaying VNC-like screenshots. The playback program is responsible for initializing and interfacing with the `CacheManager`, reading pose information from the HMD driver, and driving the display with frames obtained from the `CacheManager`.

Due to its simplicity, our playback program is wholly application agnostic; the single executable can play any memoized VR application without modification or recompilation. To switch between different VR applications, one simply redirects the program to a different set of cached files.

7. EVALUATION

We evaluate FLASHBACK's ability to improve the framerate and reduce the end-to-end latency and energy consumption of full-quality VR applications on mobile devices, using a combination of macrobenchmarks (§7.2) and microbenchmarks (§7.3).

7.1 Setup and Methodology

As mentioned in Section 6, we use the HP Pavilion 300-030 Mini as our mobile device that powers an Oculus Rift DK2 HMD. The HP Mini is a small, weak computer equipped with a mobile-class Intel HD 4400 GPU, an Intel i3 1.9GHz dual-core processor, 4GB of RAM, and a 240GB OCZ Arc 100 SSD. This setup is compatible with our existing Windows-based software stack, and as Figure 7 shows, is a suitable approximation for the state-of-the-art Samsung Galaxy S6 Gear VR mobile HMD, with HP Mini being outperformed by the Gear VR in all benchmarks. Similarly, the OCZ ARC SSD that we used presents sequential and random read speeds approximately equivalent to that of the Galaxy S6's secondary storage [33].

We evaluated FLASHBACK with Viking Village [2], a demanding virtual-world Unity application designed for high-end desktop PCs, with complex scenes that exceed the capabilities of current mobile devices. We augmented Viking Village to support virtual reality and generate megaframes; a 4K (3840x2160) megaframe allows our prototype to display 720p final frames on the HMD. Unless otherwise mentioned, all experiments are performed with two independent dynamic objects in addition to the static scene.

We measured energy consumption of our HP Mini setup, including the Oculus Rift DK2 HMD, using a P3 P4460 Kill-A-Watt Electricity Usage Monitor.

7.2 Macrobenchmarks

We evaluate FLASHBACK's real-world performance with assessments of four target objectives: *(i)* low end-to-end latency, *(ii)* high framerate, *(iii)* good visual quality, and *(iv)* low energy consumption. The following sections demonstrate that FLASHBACK compares favorably to locally rendering the VR application on both a strong desktop PC and a weak mobile device.

FLASHBACK achieves low end-to-end latency

Figure 8 demonstrates the low end-to-end latency of FLASHBACK compared to other systems. This represents the elapsed time from when the latest pose is received from the HMD until when the frame corresponding to that pose is sent to the display. It does not include the latency contributed by the input subsystem and the display hardware, as FLASHBACK has no impact on these components; those latencies are equally present when VR applications execute locally and therefore do not affect the contributions of our work.

As shown in Figure 8, FLASHBACK achieves a median end-to-end latency of 12.4ms for GPU cache hits. FLASHBACK can achieve such low latency on a GPU hit because the player pose can be sampled *right before* choosing a texture to display. Even when the requested megaframe is absent from the GPU cache and must be retrieved and decoded, FLASHBACK still incurs lower end-to-end latency than a strong desktop PC.

FLASHBACK delivers high framerates

Figure 9 presents a framerate comparison of three different VR configurations running Viking Village: local rendering on our mobile device (HP Mini), local rendering on a strong desktop PC, and FLASHBACK on the HP Mini. For a truly immersive experience, it is necessary for the system hardware running the VR application to deliver as high a framerate as the HMD supports. Figure 9 indicates that a mobile device's capabilities are insufficient to deliver a satisfactory framerate when locally rendering on a demanding VR application. In contrast, our unoptimized FLASHBACK prototype delivers a framerate 8× higher than a mobile device and even exceeds that of a desktop PC with a high-end

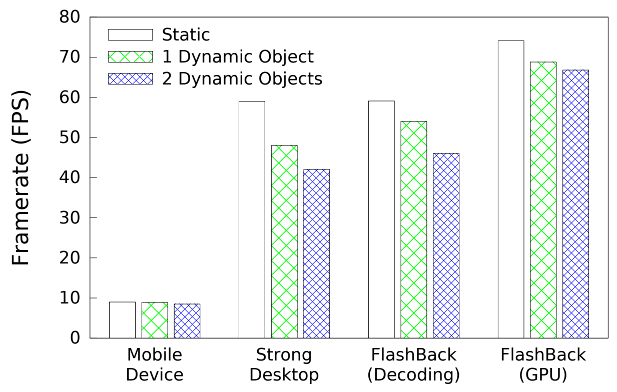


Figure 9: FLASHBACK delivers framerates up to 8X higher than a mobile HMD rendering locally.

VR implementation	Visual Quality
Standalone mobile device	0.81
FLASHBACK	0.932

Table 1: SSIM metrics showing that FLASHBACK does not negatively affect visual quality, delivering great image quality at much higher framerates. An SSIM of 1.0 indicates perfect image fidelity, and values above 0.9 are very good [12].

GPU. On a GPU hit for a static scene, FLASHBACK's framerate is limited not by our system but by the hardware refresh rate limitations of the display on the Oculus Rift DK2. We note that the performance of FLASHBACK decreases with more visible dynamic objects, which we discuss and quantify in Section 8.2.

FLASHBACK offers higher visual quality

Thus far, we have used the term *visual quality* loosely; now we provide a more rigorous definition. *Structural Similarity (SSIM)* [34] score is a standard criterion from the video compression community used to quantify the perceived loss in video quality between a pristine image f^* and a distorted version of that image, f. For example, the images f^* and f might represent the same scene rendered with a High graphics setting and a Low graphics setting, respectively. The function $SSIM_{f^*}(f)$ outputs a real value in $[0, 1]$, where a lower values indicates lower fidelity to the pristine image. By definition, $SSIM_{f^*}(f^*) = 1$ and $SSIM_{f^*}(f) = 0$ when f is random noise. While SSIM is not perfect representation of human quality assessment, it is more accurate than alternative measures such as SNR and PSNR, and maps reasonably to the standard subjective measure for video quality based on user studies, Mean Opinion Score [34].

SSIM extends from still images to video in a straightforward way: a frame-by-frame comparison of a pristine video v^* against a distorted video v, with each frame's $SSIM$ scores averaged to obtain an overall score. In our case, v^* is the pristine video generated by our high-end desktop PC tethered to the HMD, and v is the video generated and displayed by FLASHBACK.

Table 1 shows FLASHBACK's effect on Viking Village's visual quality using SSIM. We recorded a trace of head movements and replayed it on our FLASHBACK prototype to obtain v^*, and again on a stock Viking Village implementation to obtain v. Figure 10

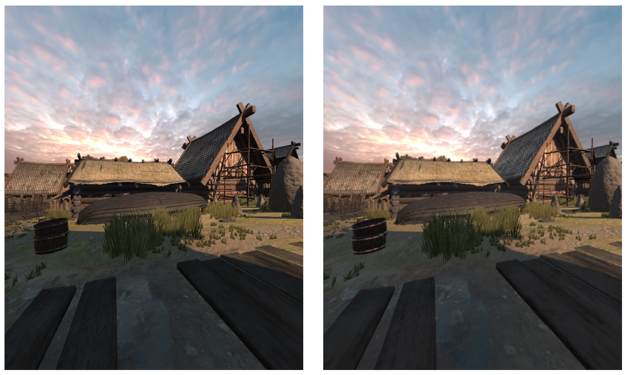


Figure 10: Sample frames outputted at the same trace pose (post-alignment), used as SSIM inputs. The left frame is the pristine image (f^*) from stock Viking Village. The right frame (f) represents a worst-case GPU cache miss in FLASHBACK, demonstrating the quality of our graphical warp.

shows a side-by-side comparison of two sample frames as outputted by Viking Village and FLASHBACK. Due to an inalterable Oculus SDK difference between Viking Village and our implementation of FLASHBACK, the images are rendered with slightly different pose values, resulting in a minor translation-only pixel misalignment. This difference stems from a Unity-specific implementation of the Oculus library that is unavailable outside of Unity and is not unique to Viking Village. We correct this alignment issue with Hugin [27], a well-known panoramic stitching tool that we configure to crop each unaligned image to the same aligned bounds. We restrict Hugin to simply shift and crop the images by a few pixels, disabling stretching, re-projection, or warping of any kind. Configured as such, Hugin does not alter the quality of either image nor unfairly affects the SSIM test results.

We found that FLASHBACK does not adversely affect visual quality, but rather achieves a higher SSIM score while delivering a much higher framerate; it also does not significantly degrade the quality as compared to stock Viking Village. Our prototype of FLASHBACK obtained a median SSIM score of 0.932 with a standard deviation of 0.015, which falls within the "great quality" threshold of the SSIM scale [34, 12]. In order to be as critical of FLASHBACK as possible, we chose to compare undistorted eye textures instead of the final frame displayed on the HMD. Eye textures are rendered at a much higher resolution than the final frame and thus represent the worst-case degradation due to FLASHBACK. On the other hand, the standalone mobile device rendering Viking Village locally obtained a poor SSIM score of 0.81, falling beneath the 0.86 threshold of "visually acceptable" [34, 12].

FLASHBACK significantly improves energy efficiency

We evaluate FLASHBACK to exemplify the energy-saving benefits of rendering memoization. Figure 11 shows that FLASHBACK consumes significantly less energy — under 250mJ per frame — than local execution on both a mobile device and desktop PC. As a frame of reference, the HP system has a minimum power consumption of 6.6W when idling with the screen on, and a maximum consumption of 28W under full utilization. Energy-efficient VR playback enables FLASHBACK to run longer on an untethered mobile HMD, providing a more immersive VR experience.

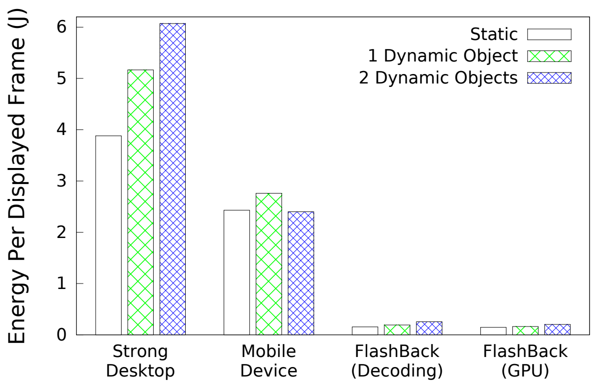


Figure 11: FLASHBACK significantly reduces the energy consumed per frame displayed even when a frame must be decoded first. Desktop measurements were conducted on a different, less energy-efficient machine.

7.3 Microbenchmarks

We now characterize the behavior of FLASHBACK with the following microbenchmarks that test query scalability to large-size caches and retrieval performance of the cache.

Cache Scalability and Performance

To determine whether the performance and scalability of FLASHBACK's frame cache satisfies the requirements of demanding VR applications like Viking Village, we designed several microbenchmarks with caches of different sizes. Figure 12a plots the median lookup time to query the cache using our universal R-tree, where all cache values are indexed at a single hierarchical level. Therefore, locating (not retrieving) a cached frame in RAM is equally as fast as locating one on disk or GPU. Figure 12a also shows that FLASHBACK's additional GPU R-tree lookup will always have a very low query time, because the on-GPU cache never exceeds a few hundred entries due to limited VRAM.

We present the cache retrieval performance from the three different sources in our current FLASHBACK implementation: the disk, system memory, and the GPU. Figure 12b shows the retrieval results for 4k megaframes. A GPU cache hit is very fast, taking less time than a vsync refresh interval. In fact, this is the reason that FLASHBACK's performance on a GPU hit is limited not by the cache itself but by the HMD's refresh rate. On the other hand, the cost of retrieving from memory and the disk is higher because the cache entry must first be decoded, the bottleneck in both cases.

Typical Cache Storage Requirements

We present in Figure 12c the cache storage size necessary to support a virtual environment of varying dimensions and complexities using our preferred quantization of 0.02 virtual-world units. As discussed in Section 4.6, the size of the static cache depends not only on the range of possible position values but also on the granularity with which we quantize the virtual environment. At this quantization granularity, while not modest, our requirements for a complex VR environment like Viking Village can fit well within the flash storage of a modern smartphone; this can be reduced significantly with selective post-deployment decompression (§4.6). In addition, while our preferred granularity is 0.02, for some users, the visual inconsistencies introduced by a granularity of up to 0.05 or 0.1 may be too small to distinguish, further reducing FLASHBACK storage requirements vastly.

8. DISCUSSION

In this section, we discuss unique aspects of FLASHBACK's behavior and analytically characterize its limitations.

8.1 Cache Hit Ratios

In FLASHBACK, the latency of delivering the most accurate megaframe for a given pose is dependent upon cache hit ratios. Note that a frame will always be displayed, but it may not be the closest match for a given pose if that frame was not yet decoded in GPU VRAM. As our evaluation has shown, the performance of FLASHBACK on a GPU hit is much higher than on misses due to decoding. While it is difficult to provide hard numbers for cache hit ratios with VR being a novelty in the consumer market, we can provide estimates. The average effective latency L_{avg} is given by the following equation:

$$L_{avg} = (hit\%) \times (L_{hit}) + (1 - hit\%) \times (L_{miss})$$

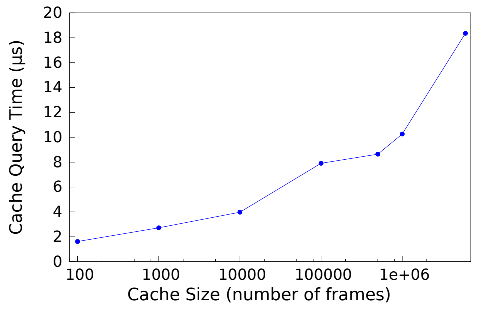
For example, a 40% GPU cache hit ratio would produce an expected latency of 34ms. Energy savings are similarly proportional to this hit ratio. In our anecdotal experience, we play-tested both a simple, single-room VR application as well as the complex Viking Village application in an informal experiment, and experienced GPU cache hit ratios from 18-47% depending on motion path and player behavior.

8.2 Scalability to Multiple Dynamic Objects

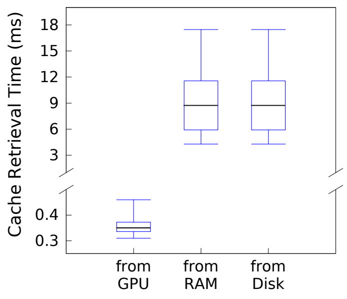
Figure 13 demonstrates that FLASHBACK can scale to compositing a moderate number of concurrently visible dynamic objects. The performance degradation is due to our unoptimized pixel shader comparing every pixel of a dynamic object's megaframe with the static scene's megaframe in the backbuffer. The current FLASHBACK prototype drastically reduces the number of pixel comparisons by merging it with the graphical warp shader, such that only pixels appearing in the *final frame* are pairwise compared instead of all pixels in the megaframe. Future improvements are possible by comparing only pixels within the dynamic object's active region instead of the entire frame.

Though the number of visible dynamic objects can be a bottleneck, this limitation warrants clarification. FLASHBACK is performance-limited by the number of *currently-visible* dynamic objects with *independent motion paths or animation characteristics*. Thus, if there are 50 instances of a rotating sprite that all rotate together or in some deterministic pattern — a rather common scenario — all 50 of those object instances can be clustered into a single dynamic object for FLASHBACK's display purposes, and quickly rendered as such. When the visible dynamic objects are truly independent, as in Figure 13, our approach is scalable only to tens of simultaneously visible dynamic objects. Anecdotally, this is a reasonable number for many virtual environments.

In the case of Figure 13, our HP setup reaches 100% physical memory usage when rendering 8 or more independent dynamic objects. This causes a memory thrashing effect where decoded textures fill GPU memory, which subsequently causes system RAM pages to be rapidly swapped to/from disk. This phenomenon occurs because our HP setup has an integrated graphics card, meaning that GPU memory is physically shared with system



(a)



(b)

VR Environment	Cache Size
Car interior	115 MB
10x10 bedroom	730 MB
20x25 lounge	2.8 GB
2500 sqft. house	8.7 GB
Basketball arena	29 GB
Viking Village [2]	54 GB

(c)

Figure 12: (a) Cache query time scales well with the size of the cache, allowing FLASHBACK to performantly support very large caches. (b) Performance when retrieving 4k megaframes from all three cache levels. The y-axis break shows very small retrieval times from GPU. (c) Cache size on Flash storage (uncompressed) for various static virtual environments.

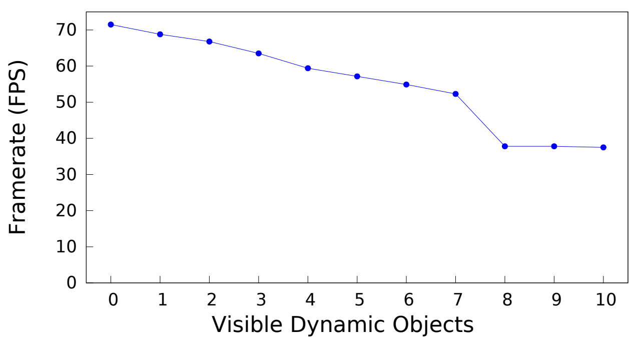


Figure 13: FLASHBACK scales well to a moderate number of visible dynamic objects, maintaining a high framerate until insufficient memory causes page thrashing.

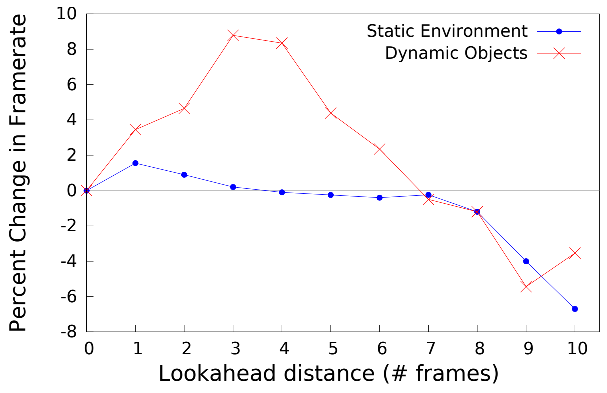


Figure 14: Speculative decoding offers some improvement for dynamic objects at moderate lookahead distances, but is ineffective when overly aggressive. All framerates are >50 FPS.

RAM, so the HP's 4GB of RAM must be split between the OS, other applications, our VR application, and FLASHBACK's cache. Even with poor memory management behavior, FLASHBACK is still able to maintain approximately 37 FPS for 10+ visible dynamic objects. Note that we did not observe this thrashing behavior on systems with dedicated graphics VRAM, as the contents of under-pressure GPU memory cannot force system RAM pages to be evicted to swap space.

8.3 Speculative Decoding Tradeoffs

As indicated in Figure 12b, decoding cached frames is among the most expensive operations on the critical display path. Fortunately, both low- and high-end mobile devices today feature an optimized hardware codec that outperforms our HP device's codec. Nevertheless, we evaluate the utility of speculative decoding to mask the decoder latency.

Figure 14 shows the effect of our simple speculative decoding method in terms of the percent framerate change for a given lookahead distance, as described in §5.4. As expected, dynamic objects are much more amenable to speculation because their poses can be easily predicted, at least when they do not interact with the player. Predicting a static frame pose is much more difficult because it depends on the player's movement, which requires a more advanced speculation algorithm to be truly effective. Our speculation turns out to be less accurate and even counter-productive when looking further ahead, as it causes frames with a higher likelihood of usage to be prematurely evicted from the GPU cache in favor of unused speculative frames.

8.4 Quantization Affects Performance

The choice of quantization granularity has an impact on FLASHBACK's performance, as seen in Figure 15. Towards the left side, coarser quantizations increase cache hit ratios, maximizing framerate and minimizing latency, but reduce visual quality because the graphical warp must stretch the image to a further perspective. Towards the right, finer quantizations maintain excellent visual quality but reduce cache locality, degrading performance.

The worst-case effect of superfine quantization is that every new pose results in a new megaframe, i.e., zero GPU cache hits. Thus, the theoretical lower bound on performance is equivalent to decoding every megaframe, shown in the third column of Figures 8 and 9. It is therefore important to select a quantization that represents the sweet spot between performance and visual quality. We have found that a quantization 0.05 virtual units (e.g., 5cm) offers playback free from noticeable visual artifacts, and that quantizations finer than 0.02 offer diminishing returns that are visually indistinguishable, hence why we chose a quantization of 0.02 when evaluating FLASHBACK.

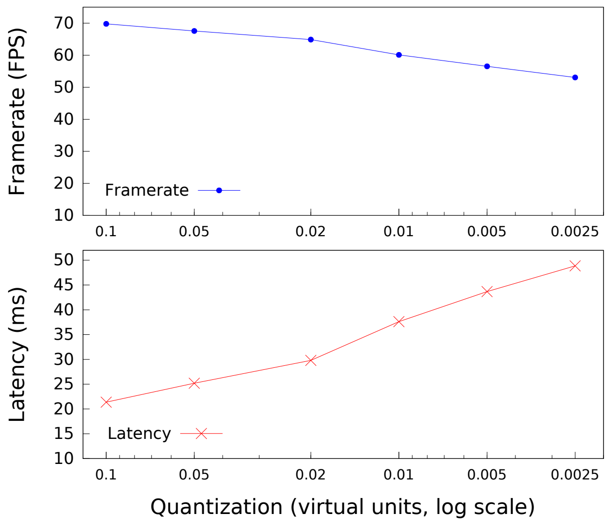


Figure 15: The effects of quantization (cache density) on FLASHBACK's performance: framerate and latency.

9. RELATED WORK

The general idea of precomputing or pre-caching results to service interactive requests on mobile devices has appeared in numerous forms [15]. For example, work has looked at precomputing statistical models for mobile appointment reminders [19], pre-caching web search results on mobile devices [23], and pre-caching database query results on mobile devices [5]. Our scenario is highly targeted at VR rendering, and therefore makes many domain-specific choices to improve graphical quality, latency, and framerate.

Instead of precomputation, an alternate way to enhance the capabilities of a mobile device is to offload computation to more powerful computers. The mobile computing community has a rich history of exploring computation offload to supplement resource-constrained devices. Among these, Maui, Clone Cloud, and Comet investigated the implications of general purpose compute offload [11, 10, 16]. In the domain of wearable visual augmented reality offload, the authors of [18] evaluated computation offload for visual and cognitive assistance tasks with Google Glass. The timing, compute, bandwidth, and quality requirements for full resolution, wide field of view, high-framerate HMDs are qualitatively much more strict than for text annotations on Google Glass, and therefore yield different points in the wearable design space. Such offloading techniques would be inappropriate for mobile VR, causing high latencies and poor visual quality.

In the space of offloading real-time rendering for mobile devices, recent work has demonstrated that it can be done with low latency [21] or high quality [12]. Specifically, Outatime [21] focuses on the case of synchronous client-server execution, in which a powerful rendering server speculates on future inputs and delivers predicted frames to the mobile client ahead of time, masking network latency. In contrast, FLASHBACK fully decouples client execution from the server — if a server is used at all — and relies instead on the mobile device's local storage hierarchy. FLASHBACK does indeed utilize a similar cube map and graphical warp IBR approximation as Outatime, albeit adapted for virtual reality. In the future, we are likely to merge these two complementary techniques into a hybrid system employing both speculation and caching to jointly improve latency and quality.

In summary, all offloading strategies, including some preliminary forays into the HMD space [13], require active, stable network connectivity to servers, ignoring the plentiful high-performance storage on the device itself.

The graphics community has examined the quality vs. performance trade-off of caching objects as rendered images. One such pioneering effort, Apple's QuickTime VR, allowed for free-viewpoint viewing of static virtual environments or objects from a desktop computer [8]. Later works helped develop algorithms to efficiently add dynamic objects into virtual environments [31, 30, 32]. However, these efforts were primarily focused on desktop environments and preceded mobile device architectures, such as: multi-level storage hierarchies, flash storage, video decoders and energy constraints. They were also less ambitious in terms of exploring the limits of memoizing dynamic object motion paths and animation sequences, standard elements of modern 3D scenes. Previous work has also proposed a cube map-like frame layout for warping pre-rendered images to the user's head pose at runtime [29, 28]. However, these latter solutions relied on specialized hardware support, termed virtual reality address recalculation pipelines, which is not present in modern GPUs. All of our work operates on commodity low end mobile GPUs.

10. CONCLUSION

We present FLASHBACK, an unorthodox solution for bringing full-quality VR experiences to weak mobile HMD devices. FLASHBACK avoids the expensive costs of a VR application's heavy rendering demands by aggressively pre-generating and caching all possible frames that a player might see. With support for caching of both static scenes and dynamic animated objects, FLASHBACK can support most VR applications in an application-agnostic fashion. We implement and evaluate a prototype of FLASHBACK to demonstrate improved energy efficiency per frame, higher overall framerates, and lower end-to-end latency compared to that of local rendering on a mobile device.

Acknowledgments

The authors thank Bernhard Reinert and Wei-ge Chen for their invaluable advice and implementation assistance. We also thank our shepherd, Youngki Lee, for his useful guidance and feedback. Kevin Boos was supported in part by NSF Award #1422312.

11. REFERENCES

[1] Minimum requirements for the Oculus DK2. https://support.oculus.com/hc/en-us/articles/201955653-Minimum-requirements-for-the-Oculus-Rift-Development-Kit-2. Accessed: 2015-11-30.

[2] Viking Village Unity Application. https://www.assetstore.unity3d.com/en/#!/content/29140. Accessed: 2015-11-30.

[3] R. Allison, L. Harris, M. Jenkin, U. Jasiobedzka, and J. Zacher. Tolerance of temporal delay in virtual environments. In *Virtual Reality, 2001. Proceedings. IEEE*, March 2001.

[4] M. Bajura, H. Fuchs, and R. Ohbuchi. Merging virtual objects with the real world: Seeing ultrasound imagery

within the patient. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92. ACM, 1992.

[5] D. Barbará and T. Imieliński. Sleepers and workaholics: Caching strategies in mobile environments. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94. ACM, 1994.

[6] D. Boman. International survey: virtual-environment research. *Computer*, Jun 1995.

[7] S. T. Bryson and S. S. Fisher. Defining, modeling, and measuring system lag in virtual environments. volume 1256, 1990.

[8] S. E. Chen. Quicktime vr: An image-based approach to virtual environment navigation. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95. ACM, 1995.

[9] D. Chu, Z. Zhang, A. Wolman, and N. D. Lane. Prime: a framework for co-located multi-device apps. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp 2015*.

[10] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11. ACM, 2011.

[11] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: Making smartphones last longer with code offload. In *MobiSys 2010*, 2010.

[12] E. Cuervo, A. Wolman, L. P. Cox, K. Lebeck, A. Razeen, S. Saroiu, and M. Musuvathi. Kahawai: High-quality mobile gaming using gpu offload. In *MobiSys*, May 2015.

[13] Y. Degtyarev, E. Cuervo, and D. Chu. Demo: Irides: Attaining quality, responsiveness and mobility for virtual reality head-mounted displays. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2015, Florence, Italy, May 19-22, 2015*, 2015.

[14] M. Deloura. *Game Programming Gems*. Charles River Media, Inc., Rockland, MA, USA, 2000.

[15] G. Forman and J. Zahorjan. The challenges of mobile computing. *Computer*, 27(4), April 1994.

[16] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. Comet: Code offload by migrating execution transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, Berkeley, CA, USA, 2012. USENIX Association.

[17] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84. ACM, 1984.

[18] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14. ACM, 2014.

[19] E. Horvitz, P. Koch, R. Sarin, J. Apacible, and M. Subramani. Bayesphone: Precomputation of context-sensitive policies for inquiry and action in mobile devices. *User Modeling 2005*, 2005.

[20] T. Jin, S. He, and Y. Liu. Towards accurate gpu power modeling for smartphones. In *Proceedings of the 2nd Workshop on Mobile Gaming*, MobiGames '15, 2015.

[21] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn. Outatime: Using speculation to enable low-latency continuous interaction for cloud gaming. MobiSys 2015, May 2015.

[22] J. Liang, C. Shaw, and M. Green. On temporal-spatial realism in the virtual reality environment. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*, UIST '91. ACM, 1991.

[23] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas. Pocketweb: Instant web browsing for mobile devices. In *Proceedings of ASPLOS 2012 (Architectural Support for Programming Languages and Operating Systems)*. ACM, March 2012.

[24] W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3d warping. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, I3D '97. ACM, 1997.

[25] Michael Abrash. What VR Could, Should, and almost certainly Will be within two years. http://media.steampowered.com/apps/abrashblog/Abrash%20Dev%20Days%202014.pdf. Accessed: 2015-11-30.

[26] Microsoft Research. Cloud-Powered Virtual Reality. http://research.microsoft.com/cloud-powered-virtual-reality/.

[27] Pablo d'Angelo. Hugin – Panorama photo stitcher. http://hugin.sourceforge.net/. Accessed: 2015-12-04.

[28] M. Regan and R. Pose. An interactive graphics display architecture. In *Virtual Reality Annual International Symposium, 1993., 1993 IEEE*, Sep 1993.

[29] M. Regan and R. Pose. Priority rendering with a virtual reality address recalculation pipeline. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94. ACM, 1994.

[30] G. Schaufler. Exploiting frame-to-frame coherence in a virtual reality system. In *Virtual Reality Annual International Symposium, 1996., Proceedings of the IEEE 1996*, Mar 1996.

[31] G. Schaufler and W. Sturzlinger. A Three Dimensional Image Cache for Virtual Reality. *Computer Graphics Forum*, 1996.

[32] J. Shade, D. Lischinski, D. H. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96. ACM, 1996.

[33] K. Vättö. OCZ ARC 100 (240 GB) SSD Review. http://www.anandtech.com/show/8407/ocz-arc-100-240gb-review/5. Accessed: 2015-12-03.

[34] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: From error visibility to structural similarity. *Trans. Img. Proc.*, 13(4), Apr. 2004.