Efficient Management of Last-level Caches in Graphics Processors for 3D Scene Rendering Workloads

Jayesh Gaur† Raghuram Srinivasan‡* Sreenivas Subramoney† Mainak Chaudhuri♯

† Intel Architecture Group, Bangalore 560103, INDIA
‡ The Ohio State University, Columbus, OH 43210, USA
♯ Indian Institute of Technology, Kanpur 208016, INDIA

ABSTRACT

Three-dimensional (3D) scene rendering is implemented in the form of a pipeline in graphics processing units (GPUs). In different stages of the pipeline, different types of data get accessed. These include, for instance, vertex, depth, stencil, render target (same as pixel color), and texture sampler data. The GPUs traditionally include small caches for vertex, render target, depth, and stencil data as well as multi-level caches for the texture sampler units. Recent introduction of reasonably large last-level caches (LLCs) shared among these data streams in discrete as well as integrated graphics hardware architectures has opened up new opportunities for improving 3D rendering. The GPUs equipped with such large LLCs can enjoy far-flung intra- and inter-stream reuses. However, there is no comprehensive study that can help graphics cache architects understand how to effectively manage a large multi-megabyte LLC shared between different 3D graphics streams.

In this paper, we characterize the intra-stream and inter-stream reuses in 52 frames captured from eight DirectX game titles and four DirectX benchmark applications spanning three different frame resolutions. Based on this characterization, we propose graphics stream-aware probabilistic caching (GSPC) that dynamically learns the reuse probabilities and accordingly manages the LLC of the GPU. Our detailed trace-driven simulation of a typical GPU equipped with 768 shader thread contexts, twelve fixed-function texture samplers, and an 8 MB 16-way LLC shows that GSPC saves up to 29.6% and on average 13.1% LLC misses across 52 frames compared to the baseline state-of-the-art two-bit dynamic re-reference interval prediction (DRRIP) policy. These savings in the LLC misses result in a speedup of up to 18.2% and on average 8.0%. On a 16 MB LLC, the average speedup achieved by GSPC further improves to 11.8% compared to DRRIP.

Categories and Subject Descriptors

B.3 [**Memory Structures**]: Design Styles

*Contributed to this work as an intern at Intel India.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
MICRO '46, December 7-11, 2013, Davis, CA, USA
Copyright 2013 ACM 978-1-4503-2638-4/13/12 ...$15.00.
http://dx.doi.org/10.1145/2540708.2540742.

General Terms

Algorithms, design, measurement, performance

Keywords

Caches, graphics processing units, 3D scene rendering

1. INTRODUCTION

High-quality and high-performance 3D scene rendering is central to the success of several important graphics applications. A general 3D rendering pipeline found in a typical GPU consists of a front-end that processes the input geometry by transforming the vertices of each primitive polygon from the local co-ordinates to the perspective view co-ordinates. The back-end of the pipeline assigns color to each pixel within each transformed primitive by possibly blending multiple render targets in the screen-space frame buffer (also known as the back buffer in DirectX[1]), applies texture maps to each pixel to bring realism to the scene, and resolves the visible pixels in the frame buffer through a depth test. Today's GPUs routinely carry out hierarchical depth tests and early depth tests to reduce depth buffer bandwidth and eliminate shading of pixels belonging to the occluded parts of a surface [12, 35, 36, 37]. Also, these processors incorporate stencil tests to apply per-pixel masks for realizing sophisticated control over the set of retained or discarded pixels in the rendering pipeline. The stencil buffer stores these masks and is often used together with the depth buffer. In summary, a 3D rendering pipeline generates access streams to different data structures such as vertices of the geometry primitives, hierarchical depth buffer (HiZ buffer), regular depth buffer (Z buffer), render targets (the pixel colors of the surfaces being rendered), texture maps, and stencil buffer.

Traditionally, the GPUs have included small independent on-die caches for each access stream type. For example, a single level of vertex and vertex index cache, Z cache, render target cache (also known as the color cache), stencil cache, HiZ cache, and multiple levels of texture caches can be found in any typical GPU. We will refer to these caches collectively as render caches. While these small render caches (few tens to few hundreds of KB) improve performance by exploiting near-term temporal locality, they fail to offer much in terms of exploiting far-flung reuses even within a frame of animation. Recent discrete and integrated graphics hardware architectures from

[1]Rendering takes place in the back buffer. A color or texture surface that needs to be rendered is referred to as a render target. When a frame is completely rendered and ready for presentation, the back buffer is swapped with the front buffer and the front buffer pixels get displayed.

Nvidia, AMD, and Intel have included reasonably large last-level caches that can be shared by all the data streams. For example, the Fermi and Kepler architectures from Nvidia respectively include 768 KB and up to 1.5 MB of shared L2 cache [34, 49]. The GPUs used in the AMD Radeon 7900, 7800, and 7700 series (Tahiti, Pitcairn, and Cape Verde of the Southern Islands family) designed based on the AMD Graphics Core Next architecture have 64 KB to 128 KB of shared read/write L2 cache attached to each memory controller channel [32]. Finally, Intel's integrated GPUs found in Sandy Bridge, Ivy Bridge, and Haswell share a large multi-megabyte LLC with the CPU cores [9, 21, 22, 39, 44, 52].

Data from different 3D graphics streams can co-exist in such a large shared graphics LLC. An efficiently-managed LLC can offer far-flung intra-stream reuses and significantly accelerate inter-stream reuses, which was impossible in the architectures with an independent cache hierarchy for each different stream. Efficient management of LLC shared among the 3D graphics streams is the central focus of this paper. This is the first study that characterizes the reuse profile of 3D scene rendering workloads and exploits the reuse behavior of different graphics data to design policies for the LLC of a GPU.

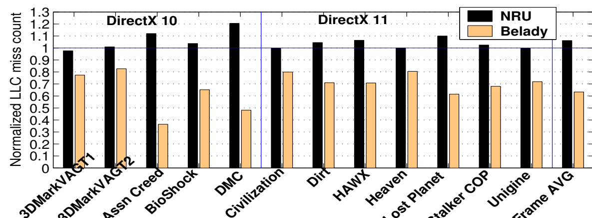

Figure 1: Number of LLC misses in twelve DirectX applications for NRU and Belady's optimal policy normalized to two-bit DRRIP in an 8 MB 16-way LLC.

To understand the potential of improving the LLC performance for 3D graphics workloads, Figure 1 shows the number of LLC misses for two different LLC replacement policies, namely, single-bit not-recently-used (NRU) and Belady's optimal [2, 33] normalized to the baseline two-bit dynamic re-reference interval prediction (DRRIP) policy [19]. The experiments are conducted on a non-inclusive/non-exclusive 8 MB 16-way LLC shared by all the graphics streams. The results are generated in an offline cache simulator, which takes as input the sequence of load/store accesses to the LLC in a typical GPU with 768 shader thread contexts (96 shader cores×8 threads per core) and twelve fixed-function (i.e., hardwired) texture samplers (one for every eight shader cores). The 3D graphics workloads consist of 52 discrete frames selected from eight DirectX games and four DirectX benchmark applications (details of the applications will be discussed in Section 2). The two-bit DRRIP policy inserts every cache block into the LLC with a re-reference prediction value (RRPV) of two or three signifying the possibility of a reuse in intermediate-future or no reuse in near-future, respectively. The choice between these two possible insertion RRPVs is made through a dynamic set-dueling technique [40]. On a hit, the RRPV of the block is made zero signifying a possibility of a reuse in immediate-future. A block with RRPV three is selected for victimization. Such a block is predicted to have no reuse in immediate- or intermediate-future. If no such block exists, the RRPV of all the blocks in the target set are incremented in steps of one until a block with RRPV three is found. Ties are broken by selecting the block with the minimum physical way id.

As shown in Figure 1, the NRU policy performs worse than DRRIP in several applications and increases the LLC miss count by 6.2% on average. Belady's optimal policy saves 36.6% of LLC misses averaged over all the frames compared to the baseline DRRIP policy pointing to the large opportunity of saving LLC misses, DRAM traffic, and system bandwidth.

Inspired by the data in Figure 1, we characterize the reuses observed within each graphics data stream as well as across different streams (Section 2). This characterization shows that Belady's optimal policy benefits from both intra-stream and inter-stream reuses. The inter-stream reuses primarily stem from the process of dynamic texture mapping where the dynamically produced render targets (i.e., color surfaces) get consumed by the texture samplers and reused for texturing other surfaces [14]. Based on this detailed characterization, we systematically derive our proposal of graphics stream-aware probabilistic caching (GSPC), which dynamically learns the reuse probability of a block belonging to a 3D graphics stream and accordingly modulates the RRPV of the block (Section 3). Our detailed trace-driven simulation results (Sections 4 and 5) show that GSPC saves 13.1% LLC misses averaged over 52 discrete frames drawn from twelve DirectX 3D rendering workloads compared to the baseline DRRIP policy. The LLC miss savings achieved by our proposal lead to an average speedup of 8.0% compared to the baseline. For a 16 MB 16-way LLC, the speedup further improves to 11.8%.

1.1 Related Work

In this section, we review the contributions on the management of LLCs in general-purpose processors and studies exploring memory management in 3D rendering hardware.

1.1.1 General-purpose LLC Management

We focus on three classes of the general-purpose LLC management algorithms that most closely relate to our proposal. We discuss algorithms for deciding the age of a block on insertion and subsequent hits in the LLC, algorithms for predicting dead blocks, and algorithms for dynamically partitioning the LLC among multiple independent threads in a multi-core setting to meet certain performance, fairness, or quality goals.

Dynamic insertion policy (DIP) adaptively inserts a block into the LLC at the least recently used (LRU) or the most recently used (MRU) position of the access recency stack depending on the outcome of a set-sampling-based duel between LRU insertion and MRU insertion policies [40]. On a cache hit, a block is always upgraded to the MRU position. The replacement policy always victimizes the block at the LRU position. This algorithm tries to eliminate the single-use blocks from the LLC as early as possible without disturbing the rest of the contents of the LLC. A subsequent proposal has shown how to employ this policy in a shared LLC of a multi-core processor so that each thread can choose the best insertion policy [20].

In a more recent work, the notions of re-reference interval prediction and re-reference prediction value (RRPV) have been proposed [19]. The age of a block in the LLC is determined based on its RRPV. If n bits are used to store the RRPV, the static re-reference interval prediction (SRRIP) algorithm statically assigns an RRPV of $2^n - 2$ to a block on insertion into the LLC. On a hit, the RRPV of the block is updated to zero. A block with RRPV $2^n - 1$ is selected as the victim. If there is no such block in the target set, the RRPV of each block in the set is incremented until the RRPV of at least one block attains a value of $2^n - 1$. The dynamic re-reference interval prediction (DRRIP) algorithm dynamically chooses between two insertion RRPVs, namely, $2^n - 2$ and $2^n - 1$ based on the outcome of a set-dueling. Thread-aware DRRIP (TA-DRRIP)

applies the technique proposed in [20] to allow multiple independent threads to execute DRRIP in a multi-core shared LLC. Recent proposals exploit signature-based hit prediction (SHiP) to improve the RRIP policies by using the program counters, memory addresses, or code path signatures of the load/store instructions [50] or extend the RRIP policies to the LLCs shared between CPU cores and a GPU running traditional GPGPU-style scientific computation workloads [28]. In contrast to these algorithms, our proposal deals with 3D scene rendering workloads and dynamically modulates the RRPV of a cache block based on the observed reuse probability of the 3D graphics stream the block belongs to. Our policy takes into account the reuse probability within and across the 3D graphics streams.

The dead block prediction algorithms correlate the program counters of the load/store instructions with the death of the cache blocks that these instructions touch [15, 23, 24, 25, 27, 29]. These algorithms victimize the predicted dead blocks early to make room for more useful blocks in the LLC. Probabilistic escape LIFO is a light-weight dead block prediction technique that does not require the program counter signature and relies only on the fill order of the cache blocks within a cache set [5]. Simple measures of dynamic reuse probability in conjunction with a clever partitioning of the address space have also been used to effectively identify the dead and live LLC blocks [4, 11]. In this paper, we apply the concept of dynamic reuse probability-based LLC management to the domain of GPUs running 3D scene rendering workloads.

Algorithms have been proposed to explicitly partition the shared LLC among the competing threads of a multi-core processor. The utility-based cache partitioning (UCP) algorithm carries out a coarse-grain partitioning of the LLC by dynamically assigning a number of ways to each thread [41]. The UCP algorithm has been extended to LLCs shared between CPU cores and a GPU where the graphics processor is employed to execute GPGPU-style workloads [28]. The promotion/insertion pseudo-partitioning (PIPP) policy improves UCP by designing smart insertion and promotion policies for cache blocks within each partition [51]. Subsequent proposals such as Vantage [42] and PriSM [31] eliminate the limitations of way-grain partitioning and allow each thread to have an arbitrary fine-grained partition. Our proposal does not carry out any explicit partitioning of the LLC among the 3D graphics streams. In fact, since the existing cache partitioning techniques do not take into account cross-thread sharing and treat the threads as independent, these techniques cannot be applied directly to the 3D graphics streams, which have significant inter-stream data sharing. We induce implicit fine-grain partitions among the streams by efficiently managing the RRPVs of the cache blocks based on the intra- and inter-stream reuses and propose for the first time an effective way of improving the LLC performance of the 3D graphics applications.

1.1.2 Memory Management in 3D Rendering

The GPUs have traditionally incorporated caching of polygon vertices (vertex cache), depth buffer (depth cache), frame or color buffer (color or render target cache), and texture (texture or sampler cache). Among these, texture caches [10] have drawn significant attention of the designers due to the high memory bandwidth consumed by the texture mapping process [3]. Texture caching is an attractive solution to reduce the texture memory bandwidth demand. The bilinear, trilinear, and anisotropic filters applied to texture stored in the form of a MIP map pyramid [48] offer significant amount of spatial and temporal locality. This observation has inspired a large number of studies on texture cache architectures including single-level texture caches [13], two-level texture caches [7], texture caches in parallel renderers [16, 47], prefetching into texture caches with deep FIFO structures [1, 17, 26, 45], victim caching [6], four-dimensional and six-dimensional tiling of texture data [13], and customized DRAM architectures for fast texture access [8, 43]. A texture cache architecture with on-the-fly decompression of texture data from a second level compressed texture cache has also been explored [45]. Reordering of ray-object intersection tests to enhance the locality of geometry and texture has been explored for a ray-tracing-based rendering engine [38].

All these studies have explored optimizations of the 3D graphics workloads and hardware to improve data locality that can be exploited by the individual render caches attached to the different 3D graphics pipeline stages. However, it is not clear how these workloads can extract the full potential of a large LLC shared between the different pipeline stages generating different streams of accesses with possibly disparate patterns. We explore solutions to this problem by understanding the memory behavior of 3D graphics workloads and incorporating this learning to develop efficient LLC management algorithms.

2. REUSE PROFILE OF 3D RENDERING WORKLOADS

In this section, we present a detailed characterization of the LLC behavior of 52 3D frame rendering jobs drawn from twelve DirectX applications. In the process, we identify the access patterns that can be potentially exploited to improve the efficiency of caching in the LLC. The details of the DirectX applications are presented in Table 1. The first column lists the applications along with the abbreviated names, if any, within parentheses. Among these, four are benchmark applications, namely, 3D Mark Vantage Graphics Tests 1 and 2 (3DMark-VAGT1 and 3DMarkVAGT2) [53], Unigine Heaven 2.1 and the Unigine 3D engine [46]. The remaining eight applications are 3D games. The second column shows the DirectX version of the applications. The last column lists the frame resolution for each application. In this paper, we consider graphics workloads built with the DirectX APIs (the Direct3D APIs within DirectX) only, since a large number of Windows PC games are designed using the DirectX APIs and these games are routinely used to benchmark the performance of the commercial GPUs.

Table 1: Details of the DirectX applications

Application	DirectX	Resolution
3D Mark Vantage GT1 (3DMarkVAGT1)	10	1920×1200
3D Mark Vantage GT2 (3DMarkVAGT2)	10	1920×1200
Assassin's Creed (Assn Creed)	10	1680×1050
BioShock	10	1920×1200
Devil May Cry 4 (DMC)	10	1680×1050
Civilization V (Civilization)	11	1920×1200
Dirt 2 (Dirt)	11	1680×1050
HAWX 2 (HAWX)	11	1920×1200
Unigine Heaven 2.1 (Heaven)	11	2560×1600
Lost Planet 2 (Lost Planet)	11	1920×1200
Stalker COP	11	1680×1050
Unigine 3D engine (Unigine)	11	1920×1200

We trace the DirectX calls generated while rendering each application frame and replay this trace of calls through a detailed in-house simulator modeling a high-end GPU. The GPU has a non-inclusive/non-exclusive 8 MB 16-way set-associative LLC shared by all the data streams. This LLC configuration

corresponds to an integrated GPU of today or a futuristic discrete GPU with a large read/write cache shared by all the graphics data to reduce the bandwidth demand on the DRAM system. A miss in the LLC always fills the requested block into the LLC and the requesting render cache. An eviction from the LLC does not send invalidation to the internal render caches of the GPU. All the results presented in this section are generated by an offline cache simulator, which has been validated against the LLC of the detailed GPU simulator. The offline model digests the LLC load/store access trace collected from the detailed simulator for each frame.

2.1 DirectX Rendering Pipeline

We begin our analysis by understanding the anatomy of the DirectX rendering pipeline and how it interacts with the graphics cache hierarchy. Figure 2 shows the organization of the DirectX 10 rendering pipeline and Figure 3 shows a high-level view of how various render caches interface with the rendering pipeline and the LLC of the GPU. The input assembler stage reads the scene geometry data such as the vertices and vertex indices from memory (through the LLC and the vertex and vertex index caches) and uses them to assemble the geometric primitives such as triangles, lines, etc.. The vertex (VTX) and vertex index (VTX index) cache misses constitute the vertex stream to the LLC. The vertex shader stage uses the programmable shader thread contexts to transform each vertex from the local co-ordinates to the world co-ordinates, from the world co-ordinates to the view space (also known as the camera or eye space) co-ordinates, and finally from the view space to the perspective projection space, which projects each vertex of the input geometry onto a projection plane within the view pyramid of the camera. The optional geometry shader stage takes the assembled primitives in the perspective projection space, creates or destroys primitives, and writes the newly created vertices to the memory hierarchy through the stream-out stage for later consumption. Since none of the rendered frames considered in this paper uses this stage, we show the vertex stream as a read-only stream to the LLC.

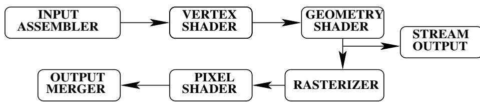

Figure 2: DirectX 10 rendering pipeline.

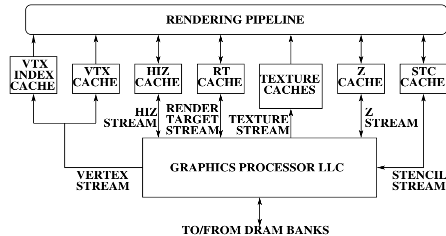

Figure 3: Graphics streams and the LLC interface.

The rasterization stage clips the primitives and the portions of the primitives that fall outside the view frustum, transforms each perspective image point to a pixel co-ordinate in the screen space or the back buffer (known as viewport transform), discards back-facing primitives (known as back-face culling), and interpolates the vertex attributes (color, normals, texture co-ordinates, depth value, etc.) of a primitive to calculate the attribute values for each pixel covering the primitive. Also, this is the stage where hierarchical and early depth tests, if enabled, are carried out to discard some of the hidden surfaces. These tests use the HiZ and Z caches and these cache misses constitute the HiZ and Z streams to the LLC. The early depth test is disabled for those render targets, pixels of which may undergo modifications to their depth values in the pixel shader.

The pixel shader stage uses the programmable shader threads to compute the color of each pixel in a render target. This is also the stage where the texture data get sampled through specific commands indicating the type of sampling (e.g., point, bilinear, trilinear, anisotropic, etc.) issued to the fixed-function texture sampler units. Certain texturing techniques such as displacement mapping (used to render realistic bumps, crevices, water waves, etc.) may require the vertex shader stage to invoke the texture sampler units. All sampler accesses go through the texture cache hierarchy and the texture cache misses constitute the read-only texture sampler stream to the LLC.

The output merger stage carries out the late depth test and stencil test to decide the final set of visible pixels. The stencil buffer accesses go through the stencil (STC) cache and the stencil cache misses constitute the stencil stream to the LLC. The output merger stage also blends the corresponding pixels of multiple non-opaque render targets to generate the final color of the pixels in the back buffer. The render target cache (RT cache) is used to hold the pixel colors of the render targets during creation, before blending, and after blending. DirectX 10 allows eight render targets to be simultaneously bound to the output merger stage and each can have blending enabled or disabled if the application needs. The RT cache misses constitute the render target stream to the LLC. The final displayable pixel color values written to the back buffer constitute the displayable color stream to the LLC.

The rendering of one complete frame may require multiple passes through this pipeline. For example, certain blended render targets can be later accessed by the samplers to use them as textures for certain surfaces within the same frame. Using a render target as a shader resource for texture sampling is usually known as render to texture [30] or dynamic texturing [14] and constitutes the primary source of inter-stream reuses (from render target production to sampler consumption) in the frames that we consider. Also, the depth buffer contents (when rendered from the light source view) can be consumed by the texture sampler during shadow mapping. However, this particular type of inter-stream reuse is not observed much in the frames that we consider. Shadow can also be implemented by creating a render target holding the depth values of the pixels viewed from each light source and reusing the render target as a texture sampler input (as in render to texture).

The DirectX 11 pipeline introduces three new stages between the vertex shader and the geometry shader to carry out hardwired tessellation of the geometry primitives. These stages are hull shader, tessellator, and domain shader.

2.2 Graphics Data Streams

This section analyzes the 3D graphics data streams and identifies the ones that have the biggest impact on the overall LLC performance. Figure 4 shows the stream-wise distribution of the accesses to the LLC. Across the board, the major fraction of the LLC traffic is contributed by the render target and texture sampler accesses. On average, these two streams constitute 40% and 34% LLC accesses, respectively. The only other stream that contributes at least 10% to the LLC accesses on average is the Z stream. The vertex and HiZ streams have 4% and 7% LLC accesses on average, respectively. The remaining 5% of the LLC accesses come from stencil, display, and other accesses such as shader code, constants, etc.. Based on this

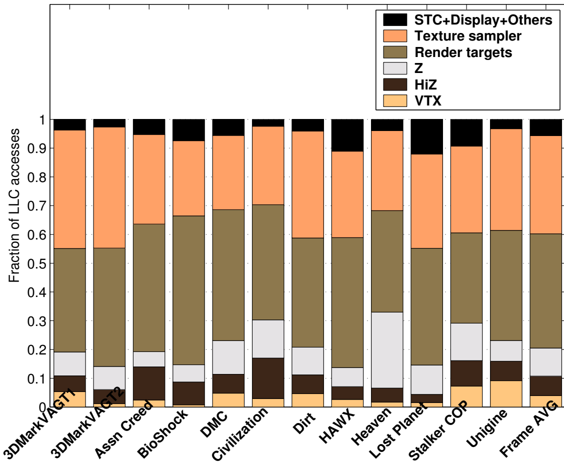

Figure 4: Stream-wise distribution of the LLC accesses in an 8 MB 16-way LLC.

distribution, in the remainder of this section, we analyze the texture sampler, render target, and Z accesses in greater detail. The display stream is the end-result of rendering of a frame, is consumed by the display driver, and does not enjoy any reuse. In Section 5, we will explore the performance benefit of not caching this stream in the LLC.

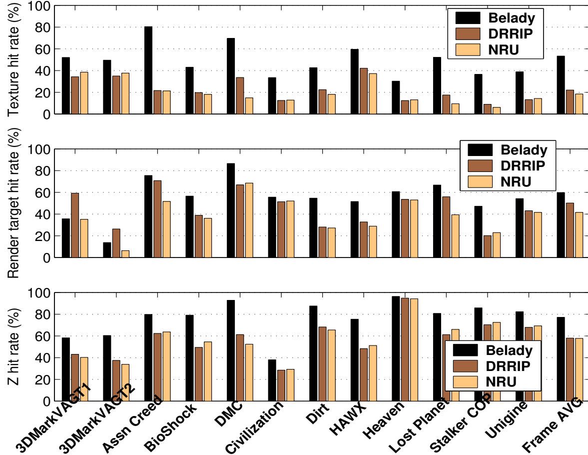

Figure 5: Hit rates for the texture sampler, render target, and Z accesses in an 8 MB 16-way LLC.

Figure 5 presents the LLC hit rates enjoyed by the texture sampler (topmost panel), render target (middle panel), and Z (bottom panel) accesses for three different policies, namely, Belady's optimal policy, DRRIP, and NRU. For the texture sampler stream, Belady's optimal policy experiences an average hit rate of 53.4%. Across the board, DRRIP and NRU deliver significantly lower hit rates averaging at 22.0% and 18.4%, respectively. On the other hand, for the render target accesses, DRRIP delivers an average hit rate (50.1%) that is within 10% of Belady's optimal (59.8% average). In this case, however, NRU lags significantly and delivers an average hit rate of only 41.5%. For the Z accesses, DRRIP and NRU perform similarly on average exhibiting a hit rate of about 58%, while Belady's optimal policy achieves a hit rate of 77.1%. From these results, it is clear that a properly managed large LLC can significantly reduce the volume of DRAM accesses in the GPUs. The largest performance improvement can come from optimizing the texture sampler accesses. In the next section, we delve deeper into the analysis of reuses enjoyed by these three streams within each frame and identify the avenues for improvement.

2.3 Inter- and Intra-stream Reuse Analysis

We first explore the nature of the reuses enjoyed by the texture sampler accesses to the LLC. Textures are either created *a priori* and not modified during rendering (static texture) or created during rendering as render targets and possibly updated in every frame (dynamic texture). Dynamic texture is the primary source of the inter-stream reuses seen in the texture sampler accesses to the LLC (from render target production to texture sampler consumption). To identify these reuses, we tag every render target block in the LLC with an RT bit. When such a block gets consumed by the texture sampler stream from the LLC, the bit is reset and the LLC hit is counted toward an inter-stream reuse. The RT bit is also reset when a render target block gets evicted from the LLC (we are not interested in tracking render target to texture reuses that do not happen in the LLC). Apart from these inter-stream reuses, both static and dynamic texture can undergo intra-stream reuses i.e., a block that does not have the RT bit set may get reused from the LLC by the texture samplers before it gets evicted.

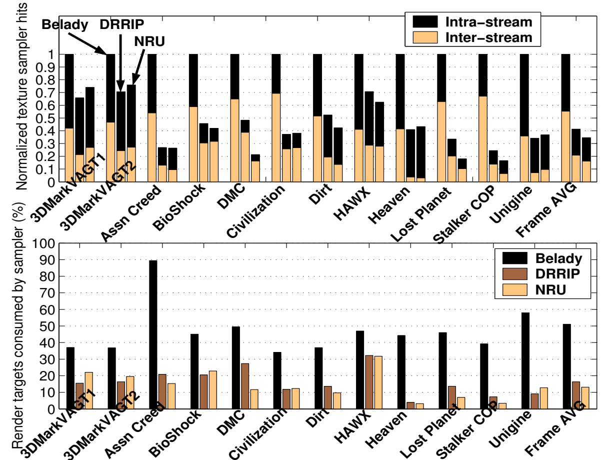

Figure 6: Upper panel: classification of the texture sampler reuses into inter-stream and intra-stream. Lower panel: percentage of the render targets consumed by the texture sampler stream through LLC hits.

The upper panel of Figure 6 shows the texture sampler hits in the LLC classified into inter- and intra-stream hits for Belady's optimal policy, DRRIP, and NRU normalized to the number of texture sampler hits enjoyed by Belady's optimal policy. On average, 55% of all texture sampler hits enjoyed by Belady's optimal policy come from inter-stream reuses, while DRRIP and NRU fall significantly short across the board. The lower panel of Figure 6 further explores these inter-stream reuses by presenting the percentage of blocks with the RT bit set that are consumed by the texture sampler from the LLC. On average, 51% of all render target blocks are consumed by the texture samplers for Belady's optimal policy, while DRRIP and NRU achieve only 16% and 13%, respectively. In Assassin's Creed, the potential render target to texture consumption rate is as high as 90%, but only a small portion of it materializes in DRRIP and NRU (21% and 15%, respectively). In summary, DRRIP and NRU are not very efficient in facilitating render target to texture consumption through the LLC. A good policy should manage the render target blocks in such a way that maximizes the texture reuses. Without specific hints from the driver, it is not possible to know which render targets will get consumed as textures in future. Therefore, our proposal will treat all render targets as potential texture sources and manage them based on the observed probability of inter-stream consumption.

Next, we analyze the intra-stream reuses in the texture sampler accesses. We divide the life of a cache block in the LLC from the time it is filled to the time it is evicted into epochs demarcated by the LLC hits the block enjoys. The epoch between hit counts k and $k+1$ will be denoted by E_k. The first epoch is E_0, which a block enters after it is filled into the LLC. Also, if a block with the RT bit set gets consumed by the texture sampler, the block becomes a texture block and enters E_0. A block currently in E_k gets promoted to E_{k+1} on observing an LLC hit. Naturally, the set of blocks in E_k is a subset of those in E_{k-1} for all $k \geq 1$. The death ratio of epoch E_k is defined as $(|E_k| - |E_{k+1}|)/|E_k|$. This is the fraction of blocks in E_k that get evicted from the LLC and fail to make it to E_{k+1}. The ratio $|E_{k+1}|/|E_k|$ will be referred to as the reuse probability of E_k. The goal of a good LLC management policy would be to assign high victimization priority to the blocks belonging to the epochs with high death ratios. In the following, we explore the death ratio of the epochs within the texture sampler stream.

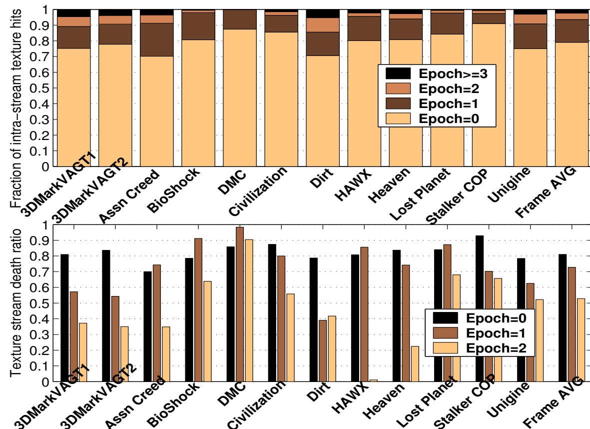

Figure 7: Upper panel: epoch-wise distribution of the intra-stream texture sampler hits. Lower panel: death ratio of each epoch of the texture blocks. The LLC executes Belady's optimal policy.

The upper panel of Figure 7 shows the epoch-wise distribution of the intra-stream texture hits when the LLC executes Belady's optimal policy. Across the board, most of the intra-stream texture sampler hits come from E_0 averaging at 79% of all texture sampler hits. A much smaller fraction (15%) of hits come from the E_1 blocks. For the E_2 and $E_{\geq 3}$ blocks, this fraction is 4% and 2%, respectively. Since this is the behavior of the optimal policy, we conclude that it is enough to keep track of the first three epochs (0, 1, and 2) of the texture blocks. These data, however, do not offer any information about the reuse probability or death ratio of the individual epochs.

The lower panel of Figure 7 presents the death ratio of each of the first three epochs of the texture blocks in the presence of Belady's optimal policy. Even though we have seen that most texture sampler hits come from the E_0 blocks, the death ratio of E_0 is extremely high averaging at 0.81. This essentially means that the reuse probability of an E_0 texture block is only 0.19. Across the board, we find that the E_0 texture blocks have very low (at most 0.3) reuse probability. The death ratio of the E_1 blocks is only slightly lower than that of the E_0 blocks averaging at 0.73. This means that the E_0 blocks that enjoy hits and move to E_1 are highly likely to get evicted before enjoying any further hits. For Assassin's Creed, BioShock, DMC, HAWX, and Lost Planet, the death ratio of the E_1 blocks is, in fact, higher than that of the E_0 blocks. Only the E_2 blocks show a reasonably high reuse probability (nearly half) with an average death ratio of 0.53 meaning that a randomly picked texture block from E_2 is almost equally likely to enjoy at least one more hit or get evicted from the LLC. In several applications, the reuse probability of the E_2 blocks is higher than half. In summary, a good policy must differentiate between the E_0 and E_1 texture blocks and handle them differently than the blocks belonging to the higher epochs. While we can assume the E_2 blocks to be mostly live, the reuse probabilities of the E_0 and E_1 texture partitions should be learned dynamically, since they vary widely across the applications, E_1 in particular.

To further understand the low texture hit rate of two-bit DRRIP, Figure 8 presents the percentage of the render target blocks and texture blocks filled by DRRIP into the LLC with RRPV equal to three. These blocks are predicted to have no reuses in the intermediate- or near-future. While DRRIP correctly learns that the texture blocks have high death ratios and inserts, on average, 36% of these blocks with RRPV=3, our analysis shows that this percentage needs to be much higher to prevent the texture blocks from thrashing the LLC. Also, on a hit to a texture block, DRRIP promotes it to RRPV zero expecting a reuse in the near-future. However, our analysis shows that the E_1 texture blocks have very low optimal reuse probability (0.27 on average). Turning to the render target blocks, we find that about one quarter of these are filled with RRPV equal to three. This may be detrimental to the inter-stream reuses because the render targets that have the potential to be consumed by the samplers may get evicted from the LLC early. Our analysis shows that under the optimal policy about half of the render targets are consumed by the sampler on average and in a few applications this fraction is much higher (lower panel of Figure 6). The render targets should be offered high protection when there is a high probability of render target consumption from the LLC by the texture samplers.

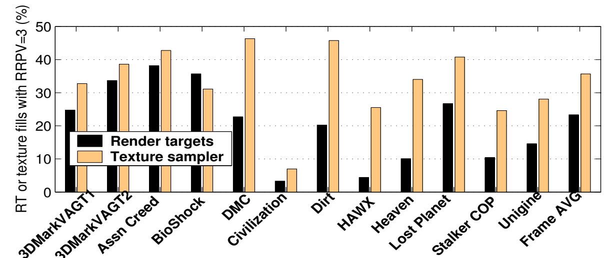

Figure 8: Percentage of the render target and texture fills with RRPV=3 in two-bit DRRIP.

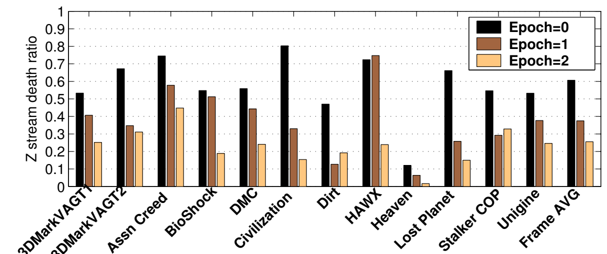

Figure 9: Death ratio of each epoch of the Z stream blocks with Belady's optimal policy.

Before closing this section, we present the epoch-wise death ratios for the Z stream blocks under Belady's optimal policy in Figure 9. The trend in the death ratio of Z blocks differs significantly from that of the texture blocks. The death ratios of the E_0, E_1, and E_2 blocks are 0.61, 0.38, and 0.26, respectively. Since only the E_0 Z blocks have relatively high death ratio, we do not keep track of the epochs for the Z stream. Instead, we

maintain the collective reuse probability experienced by all the Z blocks and use it to decide their insertion RRPV.

3. STREAM-AWARE LLC MANAGEMENT

In this section, we progressively incorporate the observations from the last section to derive three increasingly better LLC management policies for 3D graphics. We partition the LLC accesses into four streams, namely, Z, texture sampler, render targets, and the rest. Each LLC access is tagged with the identity of the source cache (Z, texture, render target, or otherwise), but we do not need to store the stream identity of any block (except the render targets) in the LLC. All our policies dedicate sixteen sets in every 1024 LLC sets to learn various reuse probabilities pertaining to the streams. These sets are identified by simple Boolean functions on the LLC index bits. We will refer to these sets as samples. The samples always execute the two-bit static re-reference interval prediction (SRRIP) policy for all the streams. In other words, a block is filled into a sample with RRPV equal to two. On a hit to a block in a sample, the RRPV of the block is updated to zero. A block with RRPV three is selected as victim with ties broken by victimizing the block with the least physical way id. In fact, in all our policies, the victim selection algorithm is the same as this. Since the samples execute the SRRIP policy, they are expected to experience reuse probabilities that are much lower than what Belady's optimal policy could have experienced. These small reuse probabilities detected in the samples must be amplified in the non-samples by controlling the RRPV of the non-sample blocks. Different reuse probability amplification techniques form the crux of our policy proposals, which we discuss next. Table 2 summarizes the activities of the LLC sample sets.

Table 2: Activities of the LLC sample sets

Activities
Updates RRPV and selects victims based on the SRRIP policy.
Learns the reuse probabilities of different graphics streams by updating a few counters on fills and hits to the sample sets. These probabilities are amplified by our policy proposals in the non-sample sets of the LLC.

Our first policy proposal incorporates rudimentary probabilistic caching for Z and texture sampler streams. Each LLC block is assumed to have an RT bit to identify the render target blocks. This bit is set on a render target access or fill and reset on a texture sampler consumption or LLC eviction. We associate four eight-bit saturating counters with each LLC bank. These will be referred to as $FILL(Z)$, $HIT(Z)$, $FILL(TEX)$, and $HIT(TEX)$. As the names suggest, the $FILL(Z)$ counter counts the number of Z stream fills into the samples. The $HIT(Z)$ counter counts the number of Z stream hits enjoyed by the samples. Similarly, we define the $FILL(TEX)$ and $HIT(TEX)$ counters. The $FILL(TEX)$ counter is also incremented when a texture sampler access gets satisfied from the LLC by a block with the RT bit set. The $HIT(TEX)$ counter is incremented when a texture sampler access gets satisfied from the LLC by a block with the RT bit reset. There is a separate seven-bit counter $ACC(ALL)$ that counts all accesses to the samples. Whenever this counter saturates, the $FILL$ and HIT counters of the Z and texture sampler streams are halved and the $ACC(ALL)$ counter is reset.

When filling a new Z block into a non-sample set of an LLC bank, the reuse probability of the Z stream is checked in that bank. If this probability is below a threshold $\frac{1}{t+1}$ i.e., $FILL(Z) > t.HIT(Z)$, the new block is filled with RRPV of three; otherwise it is filled with RRPV of two. Similarly, when filling a texture block in a non-sample set if $FILL(TEX) > t.HIT(TEX)$, the block is inserted with RRPV of three; otherwise the texture block is filled with RRPV zero because filling it with RRPV two hurts performance. All render target blocks are filled into non-samples with RRPV zero to give them the highest possible protection so that the render target to texture sampler reuses can happen through the LLC. All other blocks are filled with RRPV two. On a hit, the RRPV of the block is updated to zero irrespective of the stream. We will refer to this policy as graphics stream-aware probabilistic Z and texture caching (GSPZTC). Table 3 summarizes all the actions of the LLC controller (except halving of the counters) relevant to this policy. We discuss the influence of the parameter t in Section 5. Since we choose t to be a power of two, the RRPV computation requires only a left-shift by a constant amount followed by a comparison and a two-to-one multiplexing.

Table 3: LLC actions in the GSPZTC policy

Event	Action
Sample sets	
Z fill	$RRPV \leftarrow 2$, $FILL(Z)$++, $ACC(ALL)$++
Z hit	$RRPV \leftarrow 0$, $HIT(Z)$++, $ACC(ALL)$++
TEX fill	$RRPV \leftarrow 2$, $FILL(TEX)$++, $ACC(ALL)$++
RT → TEX hit	$RRPV \leftarrow 0$, $FILL(TEX)$++, $ACC(ALL)$++
TEX hit	$RRPV \leftarrow 0$, $HIT(TEX)$++, $ACC(ALL)$++
Other fill	$RRPV \leftarrow 2$, $ACC(ALL)$++
Other hit	$RRPV \leftarrow 0$, $ACC(ALL)$++
Non-sample sets	
Z fill	$RRPV \leftarrow (FILL(Z) > t.HIT(Z))\ ?\ 3 : 2$
TEX fill	$RRPV \leftarrow (FILL(TEX) > t.HIT(TEX))\ ?\ 3 : 0$
RT fill	$RRPV \leftarrow 0$
Other fill	$RRPV \leftarrow 2$
Any hit	$RRPV \leftarrow 0$

To compare with this policy, we derive a graphics stream-aware DRRIP (GS-DRRIP) policy from the thread-aware DRRIP policy [19], which uses set-dueling to decide between the insertion RRPVs of two and three for each of the four streams.

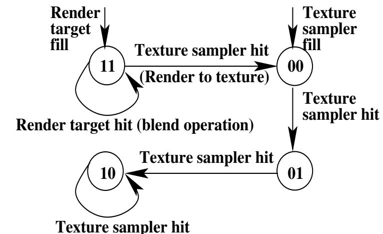

Figure 10: Two additional state bits per LLC block.

Our second policy proposal refines GSPZTC by incorporating the texture sampler epochs, while keeping everything else unchanged. To keep track of the E_0, E_1, and $E_{\geq 2}$ epochs, we incorporate two state bits with each LLC block. The epochs E_0, E_1, and $E_{\geq 2}$ are denoted by the states 00, 01, and 10. The state 11 replaces the RT bit i.e., a render target block is identified by the state 11. When a texture sampler access gets satisfied by a render target block in the LLC, its state changes from 11 to 00. Also, when a texture sampler access misses the LLC, the newly filled block is installed with state 00. Every subsequent texture sampler hit to such a block increases the state by one (done through simple logic as opposed to an incrementer) until the state reaches 10, which remains unchanged until the block is evicted from the LLC. The transitions between these states are shown in Figure 10. In a few cases, the LLC may receive a render target access to a block in the state 00, 01, or 10. Such a situation may arise if an existing render target object is reused by the DirectX application for producing a new render target. In these cases, the block transitions to

the state 11 and its RRPV is updated according to the RRPV update rule for a render target hit (same as shown in Table 3). These transitions are omitted from Figure 10 for brevity.

We replace the $FILL(TEX)$ and $HIT(TEX)$ counters by four saturating counters per LLC bank: $FILL(E, TEX)$ and $HIT(E, TEX)$ each of size eight bits, where E denotes the epoch and can take values zero and one (as already discussed, we maintain the reuse probabilities of the first two epochs only). The $FILL(E, TEX)$ counter is incremented when a texture sampler request to an LLC sample set causes the accessed block to enter the state 00 or 01 denoting epoch $E = 0$ or 1, respectively. The $HIT(E, TEX)$ counter is incremented when a texture sampler request to an LLC sample set is satisfied by a block in the LLC currently in the state 00 or 01 denoting epoch $E = 0$ or 1, respectively.

When a texture sampler access to a non-sample set of a particular LLC bank causes the accessed block to enter the state 00 (happens on a texture fill or render target to texture hit), the RRPV of the block is set to three if $FILL(0, TEX) > t.HIT(0, TEX)$ in that bank; otherwise the RRPV is set to zero. If the accessed block enters the state 01 (happens only on a texture sampler hit in the state 00), the RRPV of the block is set to three if $FILL(1, TEX) > t.HIT(1, TEX)$; otherwise the RRPV is set to zero. In all other cases, a texture sampler hit updates the RRPV of the block to zero. We will refer to this policy as graphics stream-aware probabilistic Z and texture caching with texture sampler epochs (GSPZTC+TSE). Table 4 summarizes the additional LLC controller actions relevant to this policy on top of the GSPZTC policy. In this table, we have shortened $FILL(E, TEX)$ and $HIT(E, TEX)$ to $FILL(E)$ and $HIT(E)$, respectively. The primary difference of this policy from DRRIP or GS-DRRIP (other than not resorting to set-dueling) is that on a texture sampler hit, it does not always update the RRPV to zero, but deduces the new RRPV based on the reuse probability of the new epoch.

Table 4: LLC actions for the texture sampler epochs

Event	Action
Sample sets	
RT fill/RT hit	state ← 11
TEX fill	$FILL(0)$++, state ← 00
RT → TEX hit	$FILL(0)$++, state ← 00
TEX hit	If state is 00 { $HIT(0)$++, $FILL(1)$++, state ← 01 } Else if state is 01 { $HIT(1)$++, state ← 10 } Else { state ← 10 }
Non-sample sets	
RT fill/RT hit	$RRPV$ updated as in Table 3, state ← 11
TEX fill	$RRPV \leftarrow (FILL(0) > t.HIT(0))$? 3 : 0, state ← 00
RT → TEX hit	$RRPV \leftarrow (FILL(0) > t.HIT(0))$? 3 : 0, state ← 00
TEX hit	If state is 00 { $RRPV \leftarrow (FILL(1) > t.HIT(1))$? 3 : 0, state ← 01 } Else { $RRPV \leftarrow 0$, state ← 10 }

The GSPZTC and GSPZTC+TSE policies fill the render target blocks with RRPV zero. This is done to facilitate render target to texture sampler reuses through the LLC. However, such a static policy unnecessarily increases the effective cache occupancy of the render target blocks in situations where such inter-stream reuses are unlikely. Our third policy incorporates a dynamic mechanism to manage the render target blocks on top of GSPZTC+TSE by observing the probability of consuming render targets as textures. We associate two new eight-bit saturating counters $PROD$ and $CONS$ with each LLC bank. The $PROD$ counter is incremented when a render target block is filled into an LLC sample set. The $CONS$ counter is incremented when a texture sampler access to an LLC sample set hits a block in the state 11 (recall that this state identifies a render target block). At this time the state of the block changes to 00, as already discussed in Table 4. The $CONS/PROD$ ratio is an estimate of the probability that a render target block is consumed by the texture sampler from the LLC. The $PROD$ and $CONS$ counters are halved together with the $FILL$ and HIT counters within each LLC bank.

When a render target block is filled into a non-sample set of a particular LLC bank, the inter-stream reuse probability in that bank is consulted. The new block is filled with RRPV three if $PROD > 16.CONS$ i.e., if the inter-stream reuse probability is below 1/16. The new block is filled with RRPV two if $16.CONS \geq PROD > 8.CONS$; otherwise if the render target to texture reuse probability is at least 1/8, the block is filled with RRPV zero. On a hit to a render target block from a render target access (due to blending operations), the RRPV of the block is always updated to zero.

The reuse probability thresholds (i.e., 1/16 and 1/8) need to be small because we detect these reuse probabilities from the LLC samples which execute SRRIP. The lower panel of Figure 6 shows that even DRRIP, which is expected to be better than SRRIP, has an average inter-stream reuse probability of 0.16. If our policy detects a reuse probability as small as 1/8 in the samples, it tries to amplify this probability in the non-samples by offering the highest possible protection to the render target blocks. If the detected reuse probability is smaller, our policy offers a lower level of protection to the render target blocks. We will refer to this policy as graphics stream-aware probabilistic caching (GSPC). Table 5 summarizes the new LLC controller actions (except halving of the $PROD$ and $CONS$ counters) relevant to this policy. On top of two-bit DRRIP, this policy requires two state bits per LLC block and eight eight-bit and one seven-bit saturating counters per LLC bank (two for Z, four for texture sampler, two for render target to texture, and one to count all accesses to the LLC sample sets).

Table 5: Additional LLC actions for the GSPC policy

Event	Action
Sample sets	
RT fill	state ← 11, $PROD$++
RT hit (blending)	state ← 11
RT → TEX hit	state ← 00, $CONS$++
Non-sample sets	
RT fill	state ← 11 If $PROD > 16.CONS$ then $RRPV \leftarrow 3$ Else if $16.CONS \geq PROD > 8.CONS$ then $RRPV \leftarrow 2$ Else $RRPV \leftarrow 0$
RT hit (blending)	state ← 11, $RRPV \leftarrow 0$
RT → TEX hit	state ← 00, $RRPV$ updated as in Table 4

4. SIMULATION ENVIRONMENT

We evaluate our proposal on 52 discrete frames captured from eight DirectX game titles and four DirectX benchmark applications. The details of these applications were presented in Table 1. We simulate the rendering of each frame entirely capturing several distinct phase changes that occur as rendering progresses. The DirectX calls generated while rendering each of these frames are replayed through a detailed GPU simulator.

In this paper, we focus on the GPU architectures that dedicate the entire LLC capacity to cache different graphics data, as

found in the discrete GPUs. We simulate a high-end GPU with 96 shader cores clocked at 1.6 GHz. Each core has eight thread contexts. Every cycle one SIMD instruction each from two selected threads are issued to two parallel ALU pipelines within each core. Each pipeline can execute four-wide single-precision SIMD operations including multiply-accumulates. Each core has a peak throughput of sixteen single-precision floating-point operations every cycle leading to an aggregate peak throughput of nearly 2.5 TFLOPS across 96 shader cores. The microarchitecture of each shader core resembles that of the Gen7 core used in the Intel's Ivy Bridge GPU [21], but we configure our simulated GPU such that the aggregate peak shader throughput either exceeds or closely matches that of the recent commercial discrete GPUs.[2] The shader cores share twelve texture samplers clocked at 1.6 GHz. Each sampler has a throughput of four 32-bit texels per cycle leading to a peak texture fill rate of 76.8 GTexels/second. We model a three-level texture cache hierarchy with the third level texture cache being 384 KB 48-way set-associative with 64-byte blocks. In addition to the texture cache hierarchy, we model a 1 KB 16-way vertex index cache, a 16 KB 128-way vertex cache, a 12 KB 24-way HiZ cache, a 16 KB 16-way stencil cache, a 24 KB 24-way render target cache, and a 32 KB 32-way Z cache.

The GPU has a non-inclusive/non-exclusive 8 MB 16-way set-associative LLC. This LLC capacity corresponds to a futuristic discrete GPU with a large read/write cache shared by all graphics data. We expect the LLC capacity of the GPUs to increase in future as the LLC is usually more power and bandwidth-efficient up to a certain capacity compared to the GDDRx DRAM.[3] Our default LLC configuration allows caching of all graphics data. We will also explore the impact of not caching the final displayable color data. The LLC has a block size of 64 bytes and runs at 4 GHz clock rate (modeled after the 3.9 GHz LLC of the Intel Core i7-3770 processor [18]). The round-trip load-to-use latency of an LLC bank (2 MB per bank) is minimum twenty cycles. The large, fast, banked LLC helps absorb a significant amount of DRAM bandwidth demand. We model a dual channel DDR3-1600 memory system. The DRAM part is 8-way banked with a burst length of eight and 15-15-15 (tCAS-tRCD-tRP) latency parameters.

For a four-way banked 8 MB 16-way set-associative LLC with 64-byte blocks, our GSPC policy incurs an additional overhead of 32 KB in two state bits per LLC block and 284 bits in saturating counters (see Section 3) on top of the baseline DRRIP policy. This is less than 0.5% of the LLC data array bits.

5. SIMULATION RESULTS

We evaluate our proposal in this section. First, we present a detailed performance and hardware overhead analysis in Sections 5.1, 5.2, and 5.3. Next, we evaluate our best proposal on different memory system and GPU configurations in Section 5.4 to understand its sensitivity to changed environments.

5.1 Analysis of LLC Misses

Our policy proposals have a threshold parameter t that must be fixed first. As we have already mentioned in Section 3, we expect $\frac{1}{t+1}$ to be small because this probability is detected from the SRRIP samples. Figure 11 shows the percent change in the number of LLC misses relative to $t = 16$ as t is varied. We only consider power-of-two values of t to keep the implementation simple. These data are collected for the GSPZTC policy. A positive (negative) change indicates more (less) LLC misses. While on average the four values of t experience almost the same number of LLC misses, there are visible losses in a few applications for $t = 2$ and $t = 4$. For $t = 2$, 3D Mark Vantage GT1, Dirt, HAWX, and Unigine suffer from at least 2% additional LLC misses relative to $t = 16$. For $t = 4$, Dirt and Unigine suffer from at least 1% additional LLC misses. On the other hand, Assassin's Creed delivers the best performance for $t = 2$ saving more than 4% LLC misses relative to $t = 16$. We use $t = 8$ in this paper, since it offers the most robust performance across the board.

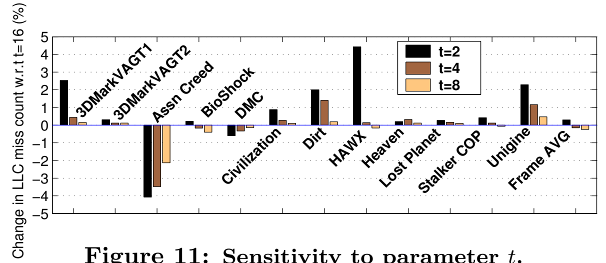

Figure 11: Sensitivity to parameter t.

Table 6: Evaluated policies

Policy	Description
DRRIP	Dynamic re-reference interval prediction
NRU	Single-bit not-recently-used
SHiP-mem	Memory signature-based hit prediction
GS-DRRIP	Graphics stream-aware DRRIP
GSPZTC	Graphics stream-aware probabilistic Z and texture caching
GSPZTC+TSE	GSPZTC with texture sampler epochs
GSPC	Graphics stream-aware probabilistic caching
GSPC+UCD	GSPC with uncached displayable color
DRRIP+UCD	DRRIP with uncached displayable color

Figure 12 presents the LLC miss count for several policies normalized to two-bit DRRIP. In addition to NRU, GS-DRRIP, and our proposals (GSPZTC, GSPZTC+TSE, GSPC), we evaluate SHiP-mem [50] and the impact of not caching the final displayable color data (GSPC+UCD and DRRIP+UCD). Table 6 summarizes all the evaluated policies. Signature-based hit prediction (SHiP) infers, at the time of filling a block in the LLC, whether the block is likely to experience future reuses. Accordingly, the newly filled block is assigned an RRPV of three (no near-future reuse) or two (possible future reuses). This inference about future reuses is carried out by learning the reuse counts observed by different types of signature associated with the cache block, e.g., program counter of the instruction that fills the block in the LLC (SHiP-PC), instruction sequence leading to the instruction that fills the block in the LLC (SHiP-Iseq), and the memory region containing the block (SHiP-mem). For graphics data, it is not possible to associate a program counter or an instruction sequence with every LLC fill because a large number of accesses come from the fixed-function hardware such as the texture samplers as well as the render target blending units and the depth test units. Therefore, we can only evaluate SHiP-mem from this family of policies. In SHiP-mem, as proposed originally, we divide the physical address space into contiguous 16 KB regions. For each region, we learn the count of reuses by hashing a 14-bit region identifier (address bits [27:14]) into a 16K-entry

[2]The aggregate peak shader throughput of our simulated GPU exceeds that of the GeForce GTX 760 and closely matches that of the GeForce GTX 780M, both based on the GK104 Kepler architecture from Nvidia.

[3]Nvidia has doubled the shared L2 cache capacity in the highest-end Kepler GPUs compared to the Fermi GPU.

table (per LLC bank) of three-bit saturating counters. On an LLC hit to a block belonging to a particular region, the corresponding region counter is incremented by one. If a block gets evicted from the LLC without experiencing any reuse, the corresponding region counter is decremented by one. A newly filled block is assigned an RRPV of three, if the corresponding region counter is zero; otherwise the block is inserted with an RRPV of two.

In Figure 12, for each application, we evaluate eight policies: NRU, SHiP-mem, GS-DRRIP, GSPZTC, GSPZTC+TSE, GSPC, GSPC+UCD, and DRRIP+UCD (from left to right). All bars are normalized to two-bit DRRIP. NRU suffers from an average increase of 6.2% in the LLC misses compared to DRRIP. SHiP-mem saves visible amounts of LLC misses in BioShock (3.8%), DMC (6.4%), Lost Planet (1.1%), and Stalker COP (2%), but, on average, experiences the same volume of LLC misses as DRRIP. In most applications, a 16 KB contiguous physical address region contains blocks from different streams and as a result, it is not possible to decipher the correct behavior of a stream by observing the collective reuse behavior of a region, as done by SHiP-mem. GS-DRRIP is able to save moderate to large amount of LLC misses with Assassin's Creed and BioShock being the biggest beneficiaries enjoying 8.5% and 5.1% less LLC misses compared to DRRIP, respectively. On average, GS-DRRIP saves 2.9% LLC misses compared to DRRIP.

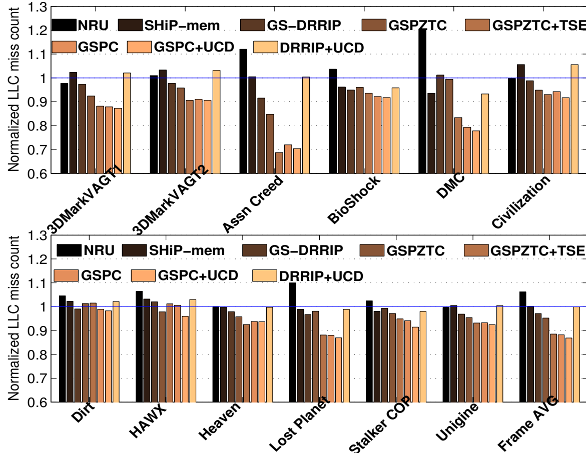

Figure 12: LLC miss count normalized to DRRIP.

Turning to our policy proposals, we observe that GSPZTC is highly effective across the board (except for DMC and Dirt) saving 4.8% LLC misses compared to DRRIP, on average. The applications that enjoy more than 5% savings in LLC misses compared to DRRIP are 3D Mark Vantage GT1 (7.6%), Assassin's Creed (15.3%), and Civilization (5.1%). There are two reasons why GSPZTC performs better than GS-DRRIP across the board (except in BioShock, Dirt, and Lost Planet). First, GSPZTC offers better protection to the render targets by filling all render target blocks with RRPV zero. Second, GS-DRRIP often fails to converge to the global optimum due to the nature of the feedback-based dueling that it uses [19, 20]. In certain situations, the dueling algorithm gets stuck in a local optimum.

Inclusion of the texture sampler epochs further saves LLC misses significantly across the board (see the GSPZTC+TSE bar). Compared to GSPZTC, the most remarkable gains are enjoyed by 3D Mark Vantage GT1 and GT2, Assassin's Creed, DMC, and Lost Planet. The applications enjoying more than 10% savings in the LLC misses with GSPZTC+TSE compared to DRRIP are 3D Mark Vantage GT1 (11.8%), 3D Mark Vantage GT2 (10%), Assassin's Creed (31.3%), DMC (16.6%), and Lost Planet (11.9%). On average, 11.5% LLC misses are saved by the GSPZTC+TSE policy compared to DRRIP.

Our final policy, GSPC, incorporates a dynamic caching algorithm for the render targets. This optimization offers less protection to the render targets in the phases where the probability of render target to texture reuses is low. However, since all of our applications heavily exploit render target to texture uses (see Figure 6), it is unlikely that this optimization will bring much benefit to this set of applications. As shown in Figure 12, a few applications benefit from this optimization, DMC and Dirt in particular (see the GSPC bar). Dirt was suffering from an increase in the LLC misses with GSPZTC+TSE, which we are able to eliminate in GSPC. In GSPC, there is no application that suffers from additional LLC misses compared to DRRIP. Assassin's Creed and DMC enjoy more than 20% savings in the LLC misses compared to DRRIP. On average, GSPC saves 11.8% LLC misses.

Uncached displayable color further improves GSPC (see the GSPC+UCD bar) saving additional LLC misses across the board, the most notable being HAWX and Stalker COP. On average, this policy saves 13.1% LLC misses compared to DRRIP. The individual application-wise savings range from 1.7% in Dirt to 29.6% in Assassin's Creed. We note that these impressive savings in the LLC misses come with a negligible bookkeeping overhead that is less than 0.5% of the total data array capacity of the LLC. Not caching the displayable color in DRRIP does not lead to any significant benefits, however (see the DRRIP+UCD bar). Only a few applications enjoy additional savings in LLC misses. These are BioShock (4.2%), DMC (6.7%), Lost Planet (1.1%), and Stalker COP (2%). On average, DRRIP with uncached displayable color experiences the same number of LLC misses as the baseline DRRIP. The primary reason why DRRIP is less sensitive to uncached displayable color is that it already inserts these blocks with RRPV three, while GSPC often fails to learn this and ends up offering the same level of protection to the display target blocks as the other render target blocks (displayable color is a render target).

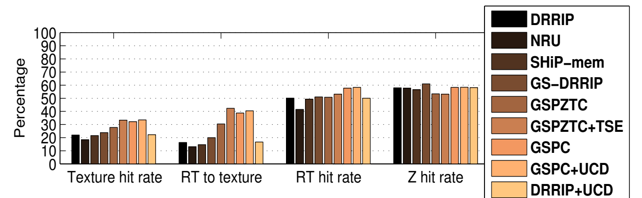

Figure 13: Texture sampler hit rate, render target production to texture consumption rate, render target hit rate, and Z hit rate.

Figure 13 further analyzes the policies in terms of the texture sampler access hit rate, percentage of the render target blocks consumed by the texture samplers from the LLC, hit rate of the render target accesses due to blending, and Z hit rate. These results are averaged over 52 frames. The texture sampler hit rate and render target to texture consumption rate gradually increase in GSPZTC and GSPZTC+TSE, as expected. Slight drops in these two dimensions are observed in GSPC due to the introduction of probabilistic render target insertion, which, due to its statistical nature, victimizes some render targets before they could be consumed by the texture samplers. However, this is compensated by uncaching the displayable color data. The render target hit rate (LLC hits enjoyed by the render target accesses originating from blending operations) increases through GSPZTC+TSE and GSPC, since these two policies gradually create more space in the LLC by eliminating the less useful

texture and render target blocks early. In fact, the average render target hit rate achieved by GSPC (57.7%) comes very close to what Belady's optimal policy achieves (59.8% as was shown in Figure 5). The Z hit rate drops visibly in GSPZTC and GSPZTC+TSE compared to GS-DRRIP due to unnecessarily high LLC occupancy of some of the render target blocks in these two policies leading to premature eviction of the Z blocks. GSPC is able to address this drawback to a great extent by evicting the less useful render targets early. GS-DRRIP, however, achieves the best Z hit rate among all the policies.

5.2 Analysis of Hardware Overhead

The GSPC policy requires four replacement state bits (two new bits in addition to the two existing RRPV bits) per LLC block. Figure 14 compares the LLC miss counts for four policies with identical replacement state bit overhead. The results are normalized to two-bit DRRIP. In the LRU, four-bit DRRIP, and four-bit GS-DRRIP policies, a block, after experiencing a hit, may take a long time to become eligible for replacement. Further, the LRU policy inserts all blocks with the highest possible protection. This particularly hurts texture management. On average, LRU suffers from a 7.2% increase in the LLC miss count compared to two-bit DRRIP. Four-bit DRRIP is only slightly better than two-bit DRRIP (0.4% on average), while four-bit GS-DRRIP saves 1.7% LLC misses compared to the two-bit DRRIP baseline. GSPC saves 11.8% LLC misses compared to the baseline, on average. These results clearly bring out the fact that GSPC makes efficient use of the additional two state bits and saves an impressive volume of LLC misses with small logic and storage overhead (less than 0.5% additional overhead per LLC block on top of baseline).

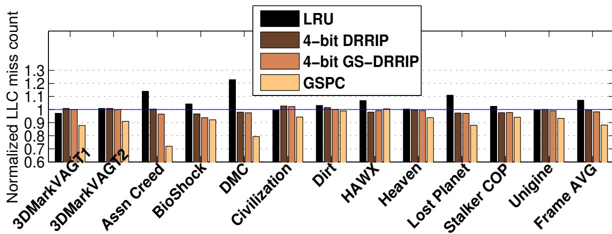

Figure 14: Comparison between iso-overhead policies.

In all our subsequent analyses, we will continue to use two-bit DRRIP and two-bit GS-DRRIP, since they offer much better cost-performance compared to their four-bit counterparts. We will also disable caching of displayable color in the LLC, but will not mention it explicitly in the policy names i.e., NRU, GS-DRRIP, GSPC, and DRRIP will stand for NRU+UCD, GS-DRRIP+UCD, GSPC+UCD, and DRRIP+UCD, respectively.

5.3 Performance Analysis

An increase in the volume of the LLC hits saves latency as well as improves the average delivered bandwidth of the memory subsystem, since the LLC is more bandwidth-efficient than the DRAM modules. In this section, we explore how the LLC miss savings translate to performance improvement. We measure the performance of each application in terms of the number of frames rendered per second. Figure 15 evaluates the performance of NRU, GS-DRRIP, and GSPC relative to DRRIP on our baseline 8 MB 16-way LLC. On average, NRU delivers 7% worse performance compared to DRRIP. GS-DRRIP fails to convert most of its LLC miss savings into performance gains. On average, GS-DRRIP improves performance by only 0.8% compared to DRRIP. It is important to note that the GPUs traditionally exploit fast thread switching to partially hide the latency of memory accesses. As a result, it is necessary to save a significantly large volume of LLC misses to achieve reasonable performance improvements in graphics applications.

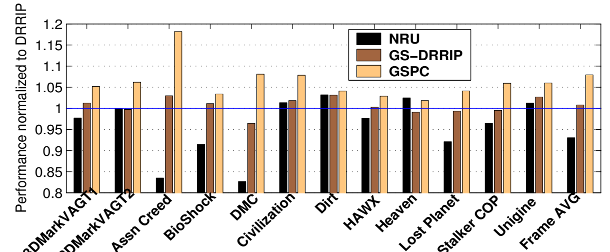

Figure 15: Normalized perf. on an 8 MB 16-way LLC.

GSPC improves performance across the board with gains ranging from 1.8% in Heaven to 18.2% in Assassin's Creed compared to DRRIP. On average, GSPC improves performance by 8% compared to DRRIP and by 16% compared to NRU. Averaged over all frames, GSPC delivers 26.1 frames per second.

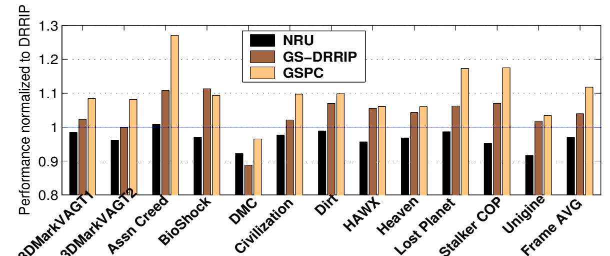

Figure 16: Normalized perf. on a 16 MB 16-way LLC.

To understand how our GSPC proposal scales to a bigger LLC, Figure 16 shows the performance of NRU, GS-DRRIP, and GSPC relative to DRRIP on a 16 MB 16-way LLC. The trends are similar to those observed in an 8 MB LLC. NRU loses 3% performance on average compared to DRRIP. It fails to offer any visible gain in any of the applications. GS-DRRIP improves performance by 4% on average compared to DRRIP. However, DMC loses by 11.2% with GS-DRRIP compared to DRRIP. Across the board, GSPC achieves impressive performance improvements compared to DRRIP. While DMC is the only application that suffers from a loss in performance (by 3.5%), some of the significant gainers are Assassin's Creed (by 27%), Lost Planet (by 17.3%), and Stalker COP (by 17.5%). On average, GSPC improves performance by 11.8% compared to DRRIP and by 15.2% compared to NRU. In absolute terms, GSPC delivers an average frame rate of 32.4 frames per second, which translates to a 24.1% improvement over its own performance on an 8 MB LLC.

5.4 Sensitivity Studies

In this section, we evaluate GSPC in two different environments, one with a faster DRAM system and another with a less aggressive GPU. In both the studies, the GPU is equipped with an 8 MB 16-way LLC. The upper panel of Figure 17 shows the performance of NRU and GSPC normalized to DRRIP for an architecture with a dual-channel eight-way banked DDR3-1867 10-10-10 DRAM system. While NRU suffers from an average performance loss of 7%, GSPC continues to improve performance across the board achieving an average speedup of 7.1% compared to DRRIP. The gains are slightly smaller compared to the slower baseline DRAM model, as expected.

The lower panel of Figure 17 presents the performance of NRU and GSPC normalized to DRRIP for a GPU with 512 shader thread contexts (64 cores× 8 threads per core) and

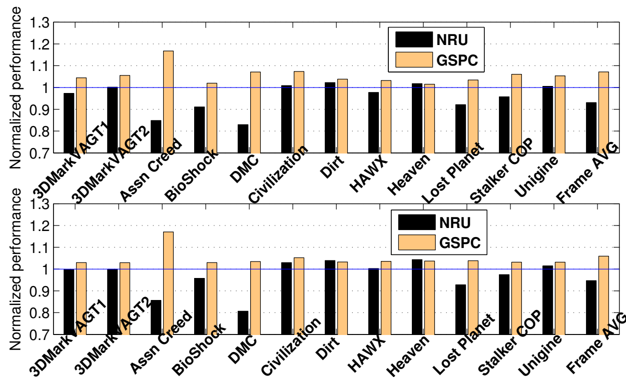

Figure 17: Perf. normalized to DRRIP for a DDR3-1867 DRAM system (upper panel) and for a less aggressive GPU (lower panel), using an 8 MB 16-way LLC.

eight texture samplers. Everything else, including the dual-channel eight-way banked DDR3-1600 15-15-15 DRAM system, is left unchanged as our baseline GPU. On this less aggressive graphics processor, NRU suffers from 5.3% loss in performance compared to DRRIP. GSPC continues to improve performance across the board achieving an average speedup of 5.9% compared to DRRIP. The less aggressive GPU has internal bottlenecks and therefore, rendering performance is expected to have less sensitivity toward memory subsystem optimizations. Overall, these sensitivity studies clearly show that GSPC is a robust algorithm that continues to deliver significant performance improvements even when the DRAM systems are made faster or in scenarios where the GPU is not very efficient. In addition to these sensitivity results, the last section has already explored the sensitivity of GSPC to the LLC capacity.

6. SUMMARY

We have presented a family of reuse probability-based last-level caching schemes for 3D graphics. This is the first study to explore caching opportunities of 3D graphics workloads in the context of multi-megabyte last-level caches. We present a detailed characterization of the rendering of 52 DirectX frames drawn from eight game applications and four benchmark applications, spanning three different resolutions and two different versions of DirectX. The characterization results show that the depth buffer values, render target colors, and texture sampling requests are the primary contributors to the last-level cache access traffic. Our proposal systematically incorporates optimizations in the caching policies to improve the volume of the last-level cache reuses enjoyed by these access streams. Our best graphics stream-aware probabilistic caching proposal achieves an average performance improvement of 8% compared to two-bit DRRIP on an 8 MB 16-way last-level cache while incurring a negligible book-keeping overhead that is less than 0.5% of the total data array capacity of the LLC. On a 16 MB 16-way last-level cache, the speedup improves further to 11.8%.

7. ACKNOWLEDGMENTS

The authors thank Rakesh Ramesh and Suresh Srinivasan for their help during the early phases of this work, and Aravindh Anantaraman, Ajay Joshi, and Supratik Majumder for useful discussions. This research effort is funded by Intel Corporation.

8. REFERENCES

[1] B. Anderson et al. Accommodating Memory Latency in a Low-cost Rasterizer. In *Proceedings of the SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 97–101, August 1997.

[2] L. A. Belady. A Study of Replacement Algorithms for a Virtual-storage Computer. In *IBM Systems Journal*, **5**(2): 78–101, 1966.

[3] E. Catmull. A Subdivision Algorithm for Computer Display of Curved Surface. *PhD thesis*, University of Utah, 1974.

[4] M. Chaudhuri et al. Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-level Caches. In *Proceedings of the 21st International Conference on Parallel Architecture and Compilation Techniques*, pages 293–304, September 2012.

[5] M. Chaudhuri. Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches. In *Proceedings of the 42nd International Symposium on Microarchitecture*, pages 401–412, December 2009.

[6] C. J. Choi et al. Performance Comparison of Various Cache Systems for Texture Mapping. In *Proceedings of the 4th International Conf. on High Perf. Computing in Asia-Pacific Region*, pages 374–379, May 2000.

[7] M. Cox, N. Bhandari, and M. Shantz. Multi-level Texture Caching for 3D Graphics Hardware. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 86–97, June/July 1998.

[8] M. F. Deering, S. A. Schlapp, and M. G. Lavelle. FBRAM: A New Form of Memory Optimized for 3D Graphics. In *Proceedings of the 21st SIGGRAPH Annual Conference on Computer Graphics and Interactive Techniques*, pages 167–174, July 1994.

[9] M. Demler. Iris Pro Takes On Discrete GPUs. In *Microprocessor Report*, September 9, 2013.

[10] M. Doggett. Texture Caches. In *IEEE Micro*, **32**(3): 136–141, May/June 2012.

[11] J. Gaur, M. Chaudhuri, and S. Subramoney. Bypass and Insertion Algorithms for Exclusive Last-level Caches. In *Proceedings of the 38th International Symposium on Computer Architecture*, pages 81–92, June 2011.

[12] N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer Visibility. In *Proceedings of the 20th SIGGRAPH Annual Conference on Computer Graphics and Interactive Techniques*, pages 231–238, August 1993.

[13] Z. S. Hakura and A. Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 108–120, May 1997.

[14] M. Harris. Dynamic Texturing. Available at http://developer.download.nvidia.com/assets/gamedev/docs/DynamicTexturing.pdf.

[15] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 209–220, May 2002.

[16] H. Igehy, M. Eldridge, and P. Hanrahan. Parallel Texture Caching. In *Proceedings of the SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 95–106, August 1999.

[17] H. Igehy, M. Eldridge, and K. Proudfoot. Prefetching in a Texture Cache Architecture. In *Proceedings of the SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 133–142, August/September 1998.

[18] Intel Core i7-3770 Processor. http://ark.intel.com/products/65719/.

[19] A. Jaleel et al. High Performance Cache Replacement using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th International Symposium on Computer Architecture*, pages 60–71, June 2010.

[20] A. Jaleel et al. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the 17th International Conference on Parallel Architecture and Compilation Techniques*, pages 208–219, October 2008.

[21] D. Kanter. Intel's Ivy Bridge Graphics Architecture. April 2012. Available at http://www.realworldtech.com/ivy-bridge-gpu/.

[22] D. Kanter. Intel's Sandy Bridge Graphics Architecture. August 2011. Available at http://www.realworldtech.com/sandy-bridge-gpu/.

[23] S. Khan, Y. Tian, and D. A. Jimènez. Dead Block Replacement and Bypass with a Sampling Predictor. In *Proceedings of the 43rd International Symposium on Microarchitecture*, pages 175–186, December 2010.

[24] S. Khan et al. Using Dead Blocks as a Virtual Victim Cache. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 489–500, September 2010.

[25] M. Kharbutli and Y. Solihin. Counter-based Cache Replacement and Bypassing Algorithms. In *IEEE Transactions on Computers*, **57**(4): 433–447, April 2008.

[26] M. J. Kilgard. Realizing OpenGL: Two Implementations of One Architecture. In *Proceedings of the SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 45–56, August 1997.

[27] A-C. Lai, C. Fide, and B. Falsafi. Dead-block Prediction & Dead-block Correlating Prefetchers. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 144–154, June/July 2001.

[28] J. Lee and H. Kim. TAP: A TLP-aware Cache Management Policy for a CPU-GPU Heterogeneous Architecture. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture*, pages 91–102, February 2012.

[29] H. Liu et al. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *Proceedings of the 41st International Symposium on Microarchitecture*, pages 222–233, November 2008.

[30] F. D. Luna. *Introduction to 3D Game Programming with DirectX 10*. Wordware Publishing Inc..

[31] R. Manikantan, K. Rajan, and R. Govindarajan. Probabilistic Shared Cache Management (PriSM). In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 428–439, June 2012.

[32] M. Mantor and M. Houston. AMD Graphic Core Next: Low Power High Performance Graphics & Parallel Compute. In *Symposium on High-Performance Graphics*, August 2011.

[33] R. L. Mattson et al. Evaluation Techniques for Storage Hierarchies. In *IBM Systems Journal*, **9**(2): 78–117, 1970.

[34] S. Molner. Design Tradeoffs in the Kepler Architecture. In *Symposium on High-Perf. Graphics*, August 2012.

[35] S. Morein. ATI Radeon HyperZ Technology. In *SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, August 2000.

[36] D. Nehab, J. Barczak, and P. V. Sander. Triangle Order Optimization for Graphics Hardware Computation Culling. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, pages 207–211, March 2006.

[37] E. Persson. Depth In-depth. Available at http://developer.amd.com/media/gpu_assets/Depth_in-depth.pdf.

[38] M. Pharr et al. Rendering Complex Scenes with Memory-coherent Ray Tracing. In *Proceedings of the 24th SIGGRAPH Annual Conference on Computer Graphics and Interactive Techniques*, pages 101–108, August 1997.

[39] T. Piazza. Intel Processor Graphics. In *Symposium on High-Performance Graphics*, August 2012.

[40] M. K. Qureshi et al. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 381–391, June 2007.

[41] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 423–432, December 2006.

[42] D. Sanchez and C. Kozyrakis. Vantage: Scalable and Efficient Fine-grain Cache Partitioning. In *Proceedings of the 38th International Symposium on Computer Architecture*, pages 57–68, June 2011.

[43] A. Schilling, G. Knittel, and W. Strasser. Texram: A Smart Memory for Texturing. In *IEEE Computer Graphics and Applications*, **16**(3): 32–41, May 1996.

[44] A. L. Shimpi. Intel Iris Pro 5200 Graphics Review: Core i7-4950HQ Tested. June 2013. Available at http://www.anandtech.com/show/6993/intel-iris-pro-5200-graphics-review-core-i74950hq-tested.

[45] J. Torborg and J. Kajiya. Talisman: Commodity Real-time 3D Graphics for the PC. In *Proceedings of the 23rd SIGGRAPH Annual Conference on Computer Graphics and Interactive Techniques*, pages 353–363, August 1996.

[46] Unigine: Real-time 3D Engine. http://unigine.com.

[47] A. Vartanian, J-L. Bechennec, and N. Drach-Temam. Evaluation of High Performance Multicache Parallel Texture Mapping. In *Proceedings of the 12th International Conference on Supercomputing*, pages 289–296, July 1998.

[48] L. Williams. Pyramidal Parametrics. In *Proceedings of the 10th SIGGRAPH Conference on Computer Graphics and Interactive Techniques*, pages 1–11, July 1983.

[49] C. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi GF100 GPU Architecture. In *IEEE Micro*, **31**(2): 50–59, March/April 2011.

[50] C-J. Wu et al. SHiP: Signature-Based Hit Predictor for High Performance Caching. In *Proceedings of the 44th International Symposium on Microarchitecture*, pages 430–441, December 2011.

[51] Y. Xie and G. H. Loh. PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 174–183, June 2009.

[52] M. Yuffe et al. A Fully Integrated Multi-CPU, GPU, and Memory Controller 32 nm Processor. In *Proceedings of the International Solid-State Circuits Conference*, pages 264–266, February 2011.

[53] 3D Mark Benchmark. http://www.3dmark.com/.