2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)

Boustrophedonic Frames: Quasi-Optimal L2 Caching for Textures in GPUs

Diya Joseph*, Juan L. Aragón†, Joan-Manuel Parcerisa* and Antonio González*
* Universitat Politècnica de Catalunya, Barcelona, Spain
† Universidad de Murcia, Murcia, Spain

***Abstract*—Literature is plentiful in works exploiting cache locality for GPUs. A majority of them explore replacement or bypassing policies. In this paper, however, we surpass this exploration by fabricating a formal proof for a no-overhead quasi-optimal caching technique for caching textures in graphics workloads. Textures make up a significant part of main memory traffic in mobile GPUs, which contributes to the total GPU energy consumption. Since texture accesses use a shared L2 cache, improving the L2 texture caching efficiency would decrease main memory traffic, thus improving energy efficiency, which is crucial for mobile GPUs. Our proposal reaches quasi-optimality by exploiting the frame-to-frame reuse of textures in graphics. We do this by traversing frames in a boustrophedonic[1] manner w.r.t. the frame-to-frame tile order. We first approximate the texture access trace to a circular trace and then forge a formal proof for our proposal being optimal for such traces. We also complement the proof with empirical data that demonstrates the quasi-optimality of our no-cost proposal.**

***Index Terms*—GPU; Caches; Graphics; Texture; Low-power;**

I. Introduction

Cache replacement policies and bypassing techniques are a common design space exploration ground in GPUs due to their high memory bandwidth usage. Although this has been extensively explored for GPGPU workloads, it has not been so for graphics workloads. In this paper, we surpass this exploration of replacement and bypassing policies by fabricating a formal proof for a no-overhead quasi-optimal caching mechanism for textures in graphics applications. With this we extend to the community a solid framework for studying texture caching (including bypassing), by skipping the exploration of non-optimal heuristics that will most likely incur a non-trivial hardware overhead.

Low-power GPUs are widely known to adopt a Tile-Based Rendering (TBR) architecture. In fact, high-end desktop GPUs are also known to adapt tiling in recent times [53]. *Tiles* are disjoint segments of the frame that have no data dependencies within themselves and can be rendered in parallel. The graphics pipeline in TBR architectures has four major types of memory accesses to main memory. They correspond to the Frame Buffer (final colors of the frame), Geometry (geometric description of the scene), Parameter Buffer (a data structure that enables the tiling of the frame), and Textures (images that enhance details of a surface of an object). Figure 1 plots the distribution of accesses to main memory among the various types of accesses in the graphics pipeline, for a set of real-world animated graphics applications that makes up our benchmark suite. We see that around 36% of accesses are texture accesses. Energy efficiency is crucial in mobile GPUs and main memory accesses constitute a significant part of the energy consumed by GPUs.

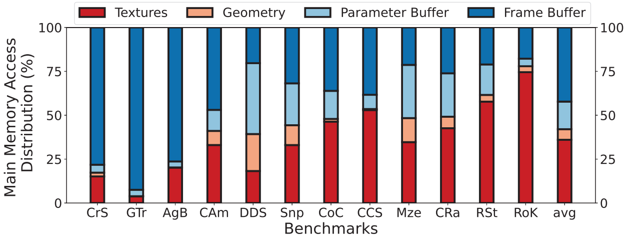

Fig. 1. Distribution of types of main memory accesses for real-world graphics applications in mobile GPUs.

Most contemporary applications that use GPUs require a high frame rate to sustain the perception of continuity while rendering realistic animations. This results in most frames being almost identical to the previous frame. This structured relationship between successive frames is known as frame-to-frame coherence. We profile our benchmark suite to find that 99% of texture accesses to the L2 are block reuses, 44% of which can be attributed to frame-to-frame coherence. This provided a strong motivation to explore the L2 texture caching efficiency.

Our benchmark suite consists of a wide variety of real-world animated graphics applications as explained in section IV. Upon profiling our benchmark suite, we observe that texture accesses follow several typical patterns:

- Frame coherence implies that a texture trace almost fully repeats itself in consecutive frames.
- We observe that, for each frame, 55% of L2 texture accesses are intra-frame reuse, 44% are inter-frame reuse, and 1% are new accesses.
- Each frame typically has a large working set resulting in large inter-frame texture reuse distances.
- The intra-frame texture reuse distances are short and can be cached efficiently with an LRU policy.

These access patterns suggest that some research is needed on how to improve L2 cache effectiveness for inter-frame

[1] Boustrophedon is a style of writing in which alternate lines of writing are reversed in order. This is in contrast to most modern languages, where the order of lines is the same, usually left-to-right.

979-8-3503-4254-3/23/$31.00 ©2023 IEEE
DOI 10.1109/PACT58117.2023.00019

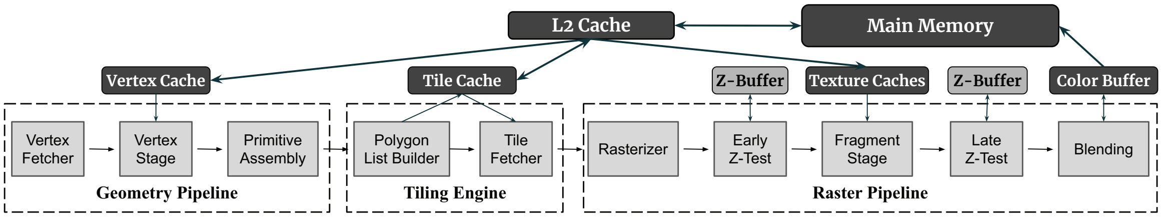

Fig. 2. The Graphics Pipeline of a TBR GPU.

reuses (without worsening intra-frame reuses). This work shows that a very simple solution is very close to optimal, and provides detailed insights and a formal proof of the reason for this quasi-optimality. This simple solution consists in reducing some inter-frame reuse distances by reversing the trace order every second frame. In other words, traversing all the frames boustrophedonically. Intuitively, the reversal starts the texture access trace with textures that were last processed in the previous frame, thus reducing some inter-frame reuse distances. Reversing the trace order is approximated by reversing the order in which the tiles of a frame are rendered in a TBR graphics pipeline. Note that tiles may be processed in any order as far as their overlapping primitives are rendered in program order.

To have a better theoretical foundation for our intuition, we first approximate our texture traces to a circular trace: one that repeats a fixed sequence of accesses to distinct memory blocks with no repetitions within the sequence. Previous works like [18], [40] have shown that LRU works very poorly with circular traces. The solution in that work was a change in the replacement policy to adapt to these traces in order to mitigate their effect on caching. However our proposal is unique to TBR GPUs and we prove that if the circularity is broken by reversing the order of the sequence in alternate iterations, LRU not only gets improved but produces the same miss ratio as OPT (an optimal replacement policy) [35] and OPTPT (an optimal placement—with bypassing—and replacement policy) [36]. Furthermore, given this additional degree of freedom to change the order of accesses in each iteration of the sequence, we also prove that LRU, with this alternating order, still gives us the minimum possible miss ratio.

To support our approximation, we first demonstrate empirically the structural similarities between our texture traces and the circular trace, w.r.t. the properties of the circular trace that make the LRU with alternate order optimal. Secondly, we demonstrate empirically, how close the tile reverse ordering approximates a pure reverse ordering of the texture access stream. Finally, we explain why reversing the tile order every other frame incurs no extra cost.

As an addition to quasi-optimality at no cost, our proposal achieves a 9.9% decrease in memory hierarchy energy.

To summarize, this paper makes the following key contributions:

- Fabricates a formal proof for the quasi-optimality of a no-cost L2 caching mechanism for textures in GPUs. We believe this formal proof of optimality is the main contribution of this paper since it proves that there is no need to investigate in alternative complex L2 cache management schemes for texture caching.
- Demonstrates the quasi-optimality empirically.
- Secondary to this, shows that this no-cost proposal translates to a 2% speedup and a 9.9% decrease in memory hierarchy energy.

The rest of this paper is organized as follows. Section II presents some background while Section III describes the proof of the quasi-optimality of our proposal. We describe the tools and workloads used to evaluate our technique in Section IV. In Section V, we present our experimental results and analysis. In Section VI, we review some related work and Section VII concludes the paper.

II. Background

Mobile GPUs typically implement a Tile-Based Rendering (TBR) architecture. The idea of TBR architectures was initially proposed to facilitate parallel rendering [11], [38]. *Tiles* are disjoint segments of the frame that can be rendered in parallel. TBR is now a common architecture adapted for low-power graphics systems where instead of tiles being rendered in parallel, they are rendered sequentially over small tile-sized on-chip buffers, which allow to exploit locality and significantly reduce power-hungry DRAM accesses and save memory bandwidth. According to a work by Antochi et al. [3], a TBR architecture reduces the total amount of external data traffic by a factor of 1.96 compared to a GPU architecture that is not tile-based (a.k.a., Immediate Mode Rendering).

A. The Graphics Pipeline

Figure 2 shows the main stages of the Graphics Pipeline and an overview of the memory hierarchy organization. In raster graphics systems, the Geometry Pipeline transforms the geometric description of a scene and creates all the *primitives* that fall inside the frustum view in accordance with the camera's viewpoint. On the other hand, the Raster Pipeline discretizes each primitive into *fragments* (at pixel granularity) that are then shaded and blended to produce the final screen image.

In a TBR architecture, the Raster Pipeline is designed to render *tiles* rather than the full frame. These tiles are usually square groups of adjacent pixels. This tiling improves locality and allows keeping on chip most bandwidth-intensive memory accesses. In order for this to happen, all the geometry needs to be sorted into subsets that will individually be able to fully render the image for each of these tiles. The process of tiling is carried out by a new pipeline stage called *Tiling Engine*.

Thus, the Graphics Pipeline for TBR architectures consists of three parts, namely the Geometry Pipeline, the Tiling Engine, and the Raster Pipeline, as shown in Figure 2.

Input data for the Graphics Pipeline consists of vertices and textures. These vertices join to form different polygons (usually triangles) called *primitives* and the textures are used to enhance details on surfaces while rendering the scene. A *Draw Command* triggers the Geometry Pipeline which fetches vertices and joins them to produce primitives. These primitives are fed as input to the Tiling Engine.

The goal of the Polygon List Builder is to produce a list (Parameter Buffer), for each tile of the screen, containing all the primitives that overlap it. After all the geometry is processed and binned, the Tile Fetcher fetches the primitives corresponding to each tile in the frame, one tile at a time. Tiles are processed in an order specified by the Tiling Engine, and their primitives are put into a FIFO queue for the Raster Pipeline to consume. Since tiles have no data dependencies among themselves, they can be processed in any order.

The Raster Pipeline renders each tile sequentially. For this purpose, the Rasterizer takes each primitive from the FIFO queue and identifies which pixels of the current tile are overlapped by the primitive. It then uses interpolation to calculate attributes for each pixel, a set of data called *fragment*. The fragments of every four adjacent pixels are grouped to form a *quad fragment* (or simply *quads*), and these quads are sent to the Early Z-Test stage. This stage uses a tile-sized buffer called the *Z-Buffer* to store the minimum depth of previously processed opaque fragments on each tile's pixel coordinate in order to eliminate those that lie behind and are invisible. The non-discarded quads are then sent to a shader core (SC), which computes an initial color for each pixel of a quad, taking into account the lighting and textures provided by the *shader program*. The output colors are then sent to the Blending Unit. This unit computes the final color of pixels, depending on the transparency of each quad, and stores them in the Color Buffer. Some rendering techniques require that the SC changes the depth of fragments, in which case the Early Z-Test is disabled and the Late Z-Test is employed. Note that both the Color Buffer and the Z-Buffer have the size of just one tile, and thus can be stored on-chip. Finally, the Color Buffer is flushed to the Frame Buffer in main memory, after a tile has been completely processed. Quads are propagated between stages through FIFO queues.

B. Memory Organization

Figure 2 illustrates that there are multiple L1 caches for different data structures used by different parts of the pipeline, backed by a shared L2 cache, which is ultimately backed up by main memory.

C. Tile Orders

It is imperative to this paper to understand that tiles can be processed in any order throughout the frame as long as the primitives in the frame, and thus within each of these tiles, are processed in program order. Intuitively, although tiles have no data dependencies among themselves, the Parameter Buffer and texture accesses do possess locality among tiles. Typically, spatially closer tiles have more locality in the Parameter Buffer as primitives have a higher chance of overlapping tiles that are close together. Textures, on the other hand, have spatial and temporal locality at the borders of the tiles. That is, adjacent tiles with shared edges are likely to access the same texture blocks of memory. Textures may also repeat in random locations throughout the frame (e.g., trees that have the same texture) and thus some tiles that are far away may also have locality.

The most well-known orders in graphics are Scan-line and Z-order. The Z-order is quite popular in computer graphics and image processing due to certain properties that enhance locality in memory accesses and its ease of implementation. In this work, we use the Z-order for traversing tiles within a frame.

D. OPT and OPTPT

We prove the optimality for the approximated memory reference trace by comparing formally derived observations of an optimal replacement policy (OPT) and an optimal placement (bypassing) and replacement policy (OPTPT). The OPT replacement policy was formally proved optimal by Mattson et al. [35]. Conceptually, OPT requires a look into future memory accesses in order to make a decision on replacement. In particular, upon a cache miss, it replaces the line in the corresponding set that will be accessed the farthest away in the future. On the other hand, OPTPT was proved optimal by McFarling [36]. This policy is the optimal replacement and placement policy for a cache that allows bypassing. On a cache miss, it compares the next access of all the memory blocks in the corresponding set as well as the memory block that caused a miss. In case there is any block in the set that will be next accessed farther away in time than the block that caused a miss, then a normal OPT replacement is performed. In case the block that caused a miss has the next access farther than any of the blocks already in the set, then the block is bypassed in that cache. Both of these policies require perfect knowledge of future accesses and are thus deemed infeasible. Yet they have helped evaluate the efficacy of caching policies all through literature.

III. Quasi-Optimal Texture Caching

Figures 1 and 11 show that, on average, textures contribute towards 33.7% of main memory accesses and 63.6% of the last level (L2) cache misses. The main contribution of this work is identifying a dominant pattern in texture memory accesses to the L2 cache across multiple frames and proposing and justifying a no-overhead, quasi-optimal technique to exploit reuse in texture accesses to the L2 that reduces costly main memory accesses, thus reducing energy consumption and increasing performance. Note that energy consumption is critical in mobile GPUs and performance is crucial for real-time rendering applications.

The main challenges with exploring the texture access reuse in the L2 and proposing quasi-optimal caching and proving it quasi-optimal are the following.

- Identifying the dominant pattern for texture accesses in a shared L2 cache.
- Discovering both short-term and long-term reuse distances in the pattern and deciding how to maximally exploit both types.
- Given the added degree of freedom of the decision of tile order in each frame, proposing a no-cost optimal caching solution with this added freedom.

A. Our Approach

Upon profiling our benchmark suite, we observe that texture accesses follow several typical patterns:

- Frame-to-frame coherence implies that a texture trace almost fully repeats itself in consecutive frames.
- We observe that, for each frame, 55% of L2 texture accesses are intra-frame reuse, 44% are inter-frame reuse, and 1% are new accesses.
- Each frame typically has a large working set resulting in large inter-frame texture reuse distances.
- The intra-frame texture reuse distances are short and can be cached efficiently with an LRU policy and contemporary L2 associativities.

We choose to preserve the caching efficiency provided by LRU for intra-frame reuses and instead focus on caching inter-frame reuses by exploring the design space opened up by the freedom to choose the tile order in each frame of a TBR architecture. We propose using a *boustrophedonic* tile order: i.e., to reverse the tile order in every second frame (that approximates the reversal of the texture access stream). We dub this combination of LRU with a reverse order in every alternate frame as LRU_RO and the baseline LRU with the conventional forward order in all frames as LRU_FO. We also use the suffixes $\_RO$ and $\_FO$ for the rest of the replacement policies that we use for comparison. It is important to note that the tile order within a frame remains unchanged as the baseline Z-order. As part of our analysis, we approximate the texture trace to a circular trace to investigate the potential for inter-frame caching and formally prove that LRU_RO is optimal for such an approximated trace. We then support our approximations with empirical data showcasing the structural similarities between our texture traces and the circular trace, w.r.t. the properties of the circular trace that make LRU_RO optimal. Secondly, we demonstrate empirically how closely the tile reversal order approximates a pure reverse ordering of the texture access stream in a frame. We end the section by providing further insights into texture caching.

ABCDEFG
ABCDEFG
ABCDEFG
ABCDEFG
•••

Fig. 3. A circular trace with seven distinct accesses.

B. Optimality Proof

We approximate the texture access trace of the L2 cache, along multiple frames, to a circular trace, to formally analyze caching for inter-frame reuses. A circular trace, as explained before, is a trace where a loop of $'s'$ number of distinct memory blocks (in a fixed order) are repeatedly accessed infinitely. Figure 3 shows an example of a circular trace. A circular trace is known to defeat the LRU replacement policy in a fully associative cache of size $'j'$, if $s > j$. This implies that the miss ratio in such a case would be 1 (no hits). At the same time, a circular trace is also known to work quite poorly for the OPT replacement policy if $s > j$. Note that all proofs and observations, henceforth, for fully-associative caches can be extended for set-associative caches, as each set in the cache behaves like an independent fully-associative cache of the size equivalent to the associativity of the cache.

A previous work [37] formally calculates the Steady State Miss Ratio (SSMR) for a circular trace with the optimal replacement policy (OPT), as follows. As before, the number of distinct memory block accesses in the loop is s and the size of the cache is j.

$$SSMR = \frac{s-j}{s-1} \quad \forall \ s > j$$

If we were to go a step further and estimate the benefits of an optimal replacement policy with optimal bypassing, we would use the OPTPT (OPT Pass-Through) bypassing policy, formally proved optimal by McFarling [36]. According to the proof in this paper, the SSMR for OPTPT for a circular trace is as follows.

$$SSMR = \frac{s-j}{s} \quad \forall \ s > j$$

In Section II, we explained that tiles may be processed in any order as long as their overlapping primitives are rendered in program order. Assuming, for now, that this extends to the texture access order, we introduce a degree of freedom to our approximated circular trace. We define a new trace pattern called *loop traces* where a loop of s number of distinct memory blocks (in any order) are repeatedly accessed infinitely. This implies that each iteration could have any order allowed by the permutations of s blocks. Note that both circular traces and “alternate reverse order” traces are a subset of loop traces. In this subsection, we first put a lower bound to the minimum possible miss ratio for a loop trace given any replacement policy, bypassing and order of accesses in each iteration. We then calculate the miss ratio for LRU_RO in an “alternate reverse order” trace and find it equal to the lower bound in a loop trace.

1) Theorem 1: For a fully associative cache of size j, given that a set of distinct accesses s has to repeat infinitely in a loop form and we have control over the order of the accesses for each iteration (i.e., it is a loop trace), the maximum number of hits in each iteration (other than the first) is j, $\forall\ s > j$ since this is the maximum number of blocks that can be stored in the cache; or it is s, $\forall\ s <= j$ since this is the number of blocks in each iteration. Therefore, the lowest possible miss ratio after infinite M iterations (i.e., the lowest SSMR) is given as follows.

$$SSMR >= \lim_{M\to\infty} \frac{s+(M-1)*(s-j)}{M*s} \quad \forall\ s>j$$

$\Rightarrow$

$$SSMR >= \frac{s-j}{s} \quad \forall\ s>j$$

2) Theorem 2: For a fully associative cache of size j, given that a set of distinct accesses has to repeat in a loop form and we have control over the order of the accesses for each iteration, the SSMR for LRU_RO is calculated as follows.

For every iteration other than the first iteration, the number of hits is again exactly equal to the size of the cache. This is because the state of the LRU stack after a finished iteration has, from top to bottom, the exact same order as the order of blocks accessed in the next iteration (reverse order of previous iteration). Thus, the first j accesses will be hits. Therefore, if the loop continues for M cycles, the number of misses would be $s+(M-1)*(s-j)$ and the number of accesses would be $M*s$. Therefore, the SSMR is the same as in Theorem 1.

$$SSMR = \frac{s-j}{s} \quad \forall\ s>j$$

Theorem 1 delivers a lower bound for the SSMR for loop traces but does not prove if there exists a replacement or placement policy that can achieve this SSMR. Theorem 2 shows us that LRU_RO reaches this lower bound. This automatically implies that OPT_RO and $OPTPT\_RO$ will also reach the lower bound as they have been proven to be optimal.

We conclude from here that, for a loop trace where $s > j$, the minimum SSMR is $\frac{s-j}{s}$. For a circular trace, LRU has a miss ratio equal to 1, OPT has a miss ratio equal to $\frac{s-j}{s-1}$, and OPTPT has the lowest miss ratio equal to $\frac{s-j}{s}$. On the other hand, for the alternate reverse order, all three, LRU, OPT and OPTPT, have the SSMR equal to $\frac{s-j}{s}$, the lowest possible SSMR for loop traces.

C. Interesting Derivations

To better visualize the theorems and the above analysis, let's look at Figure 4. It shows the misses, hits and bypasses for LRU, OPT and OPTPT with both circular and alternate reverse orders for an example loop trace with seven distinct accesses (A, B, C, D, E, F, G), assuming a cache size of 3. For the circular order, we clearly see that 1) LRU gives no hits; 2) OPT and MRU behave similarly; and 3) OPTPT achieves the optimal miss ratio by bypassing almost all misses (seen in black). For the alternate reverse order, we see the same behavior from all three LRU, OPT and OPTPT. Note here that OPTPT in alternate reverse order does not bypass a single access and still achieves the minimum possible SSMR. This implies that the characteristic of reversing the order itself has led to the optimal hits and does not require bypassing anymore. This is because the OPT stack and the OPTPT stack are in the same state as the LRU stack in this loop order. This observation also demands the following discussion on the triumph of LRU_RO over bypassing.

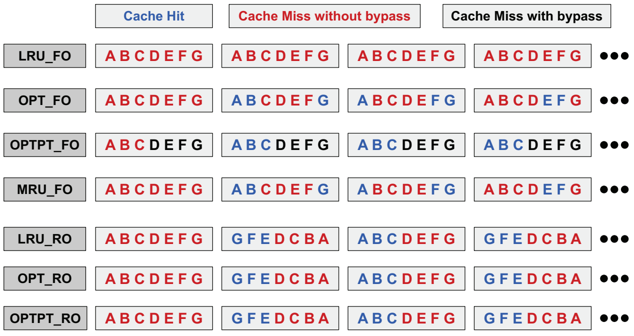
Cache Hit
Cache Miss without bypass
Cache Miss with bypass
LRU_FO A B C D E F G A B C D E F G A B C D E F G A B C D E F G •••
OPT_FO A B C D E F G A B C D E F G A B C D E F G A B C D E F G •••
OPTPT_FO A B C D E F G A B C D E F G A B C D E F G A B C D E F G •••
MRU_FO A B C D E F G A B C D E F G A B C D E F G A B C D E F G •••
LRU_RO A B C D E F G G F E D C B A A B C D E F G G F E D C B A •••
OPT_RO A B C D E F G G F E D C B A A B C D E F G G F E D C B A •••
OPTPT_RO A B C D E F G G F E D C B A A B C D E F G G F E D C B A •••

Fig. 4. Circular and "alternate reverse order" traces with LRU, OPT, OPTPT and MRU in a cache of size 3.

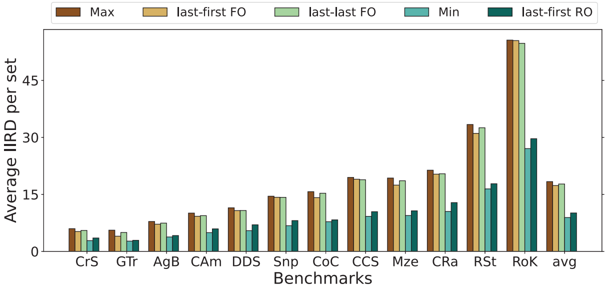

Fig. 5. IIRD (Max, Min, last-last, last-first) of FO and RO.

Caches are often bypassed if it is known that the memory blocks that are going to be fetched will not have any reuse in the future or the reuse distance is so large that the capacity of the cache forces an eviction before such a reuse could take place. Bypassing requires heuristics to know which memory blocks to bypass. Wrong bypassing can easily lead to worse caching. Note that heuristics usually have hardware overheads. The proof above shows how LRU with reverse order is as good as perfect bypassing, for loop traces. Thus, it seems that LRU_RO has essentially bypassed cache bypassing for loop traces, provided the aforementioned degree of freedom to change orders.

Note that the fact that MRU_FO behaves similar to OPT_FO indicates that MRU_FO should also behave better than the baseline (LRU_FO). We shall explore this in Section V-C.

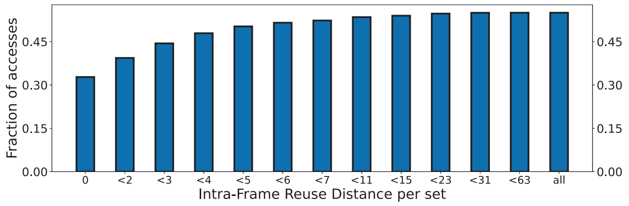

Fig. 6. Distribution of intra-frame reuse distances within each set averaged over the benchmark suite.

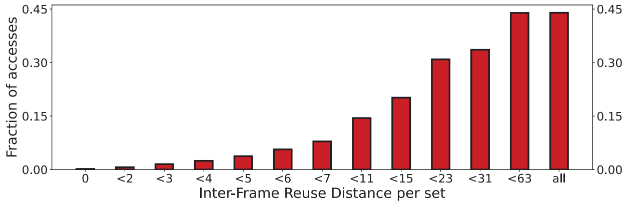

Fig. 7. Distribution of inter-frame reuse distances within each set averaged over the benchmark suite.

D. Approximation of Texture Traces

In this subsection we first approximate texture traces to a loop trace and then approximate it to a circular trace. We also show that upon the pure reversal of these texture traces in every second frame, these traces approximate to an 'alternate reverse order' trace. Note that these approximations are only valid when using LRU, OPT or OPTPT policies.

As mentioned before, there are two major characteristics of a loop trace. One is that this trace is a set of distinct accesses that keeps repeating in every iteration of an infinite loop. The second is that there are no repetitions of accesses within each iteration. For texture traces, loop iterations correspond to frames. Let's first explore the frame-to-frame coherence for textures, i.e., if a set of distinct accesses repeats itself every iteration.

Figure 3 shows that for a circular trace with s distinct accesses, each access will have an inter-iteration reuse distance of $s-1$. Introducing an extra access in the second iteration, the reuse distance would increase by 1 and thus be s. Similarly, in the case of a deletion, the inter-iteration reuse-distance (IIRD) would be $s-2$. Generalizing,

$$IIRD \quad = \quad (s \ - \ 1) \ + \ Insertions \ - \ Deletions$$

Figure 5 shows the average inter-frame reuse distance (IIRD for texture traces) in yellow (labeled last-first FO) and the average distinct accesses per frame minus 1 (s-1) per set in brown (labeled Max), over 30 frames for all the benchmarks in our benchmark suite. We see that on average both the values are almost equal, indicating that insertions and deletions in the texture trace between two consecutive frames must be equal. Empirical data tells us that insertions (new texture blocks accessed) amount to only 1% of accesses in every frame. This implies that deletions must be as few as insertions. Thus, the set of distinct accesses in every frame is approximately the same.

Figure 6 plots the fraction of texture accesses that have an intra-frame reuse distance (within each set, for a cache with 2048 sets) corresponding to the bins in the x-axis, for 30 frames, averaged over all applications in our benchmark suite. Figure 7 plots the same for the fraction of inter-frame reuse distances. It can be seen that around 55% of accesses are intra-frame reuses thus 55% are repetitions within an iteration. And yet on careful observation, we note that around 31% of these have a reuse distance of zero, meaning that they are consecutive accesses to the same block. These will be hits for all replacement policies and most placement policies. Thus we only need to worry about the rest of the 24% of the accesses. These 24% violate the second characteristic of loop traces.

But, when talking about inter-frame caching in our texture trace with an LRU_RO or LRU_FO, what matters for the LRU or OPT stack at the beginning of a frame is the sub-trace of the last access to each block in the previous frame. As for the current frame, the accesses that will benefit from inter-frame reuses are the first accesses to each block. The rest of the accesses in the current frame will be intra-frame reuses and we see in Figure 6, that these have small reuse distances and can be cached easily with an LRU with conventional associativities. Michaud [37] also proves that for a given trace, all LRU hits are OPT hits, therefore these intra-frame reuses will also be cached by the OPT. Note that the shortness of intra-frame reuses are not an accident and can be attributed to the intra-frame tile order, Z-order, that preserves texture locality among adjacent tiles.

So, to prove that texture traces are approximately circular traces (w.r.t. the properties of LRU_FO, OPT_FO and $OPTPT\_FO$), we prove that the sub-trace of last accesses to all blocks in the previous frame and the sub-trace of first accesses to all blocks in the current frame (same tile order) are identical. Similarly, to prove that upon fully reversing the texture trace for every other frame, the texture traces become approximately "alternate reverse order" traces (w.r.t. the properties of LRU_RO, OPT_RO and $OPTPT\_RO$), we would prove that the sub-trace of last accesses to all blocks in the previous frame and the sub-trace of last accesses to all blocks in the next frame (same tile order) are identical. These last accesses of the second frame will become the first accesses when this frame uses a reverse tile order.

There are many metrics to quantify the similarity between two sequences or traces (in this case). Since we deal mostly with reuse distances, we propose a metric that uses reuse distance to quantify the similarity. Given a loop trace with s distinct accesses, the maximum average inter-iteration reuse distance (IIRD) is $(s-1)$. This is because there are only s distinct accesses, so the maximum possible IIRD for each reuse can only be $(s-1)$. The case where each reuse has this maximum reuse distance $(s-1)$ happens only when the loop trace is a circular trace. On the other hand, the minimum sum of IIRDs in a loop trace is $0+1+2...+(s-1) = ((s-1)*s)/2$ and this is possible only when the loop trace is an alternate

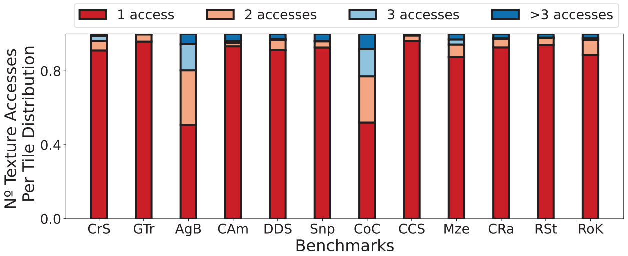

Fig. 8. Distribution of the number of L2 texture accesses per set per tile.

reverse order trace. This implies that the minimum average IIRD for a loop trace is $(s-1)/2$. In summary, for loop traces, IIRD ranges between $(s-1)/2$ and $(s-1)$. The closer it is to the maximum, the closer the sub-trace is to a circular trace. The closer it is to the minimum, the closer the sub-trace is to an alternate reverse order trace.

We collect last access sub-traces and first access sub-traces for all frames in a simulation with our baseline tile order for frames (forward order). In Figure 5 we plot the maximum IIRD, the average IIRD between the last access sub-trace of a frame and the first access sub-trace of the next frame (labeled as last-first FO), and the average IIRD between the last access sub-trace of a frame and the last access sub-trace of the next frame (labeled as last-last FO). We find the first two values to be very close to each other and thus our texture traces are approximately circular traces for LRU_FO, OPT_FO and $OPTPT\_FO$. We also find the first and third to be almost equal and thus our texture traces (when purely reversed for every second frame) are approximately "alternate reverse order" traces for LRU_RO, OPT_RO and $OPTPT\_RO$.

Note that pure reversal of a texture trace in a frame is not possible (further discussed in the next subsection). In Figure 5 we also plot the minimum IIRD and the average IIRD between the last access sub-trace of a frame and the first access sub-trace of the next frame from simulations done with alternate reverse tile order in frames. We find this IIRD to be very close to the minimum with a slight error. This error can be attributed to the fact that we do a tile reversal and not a full reversal of the trace.

E. Reverse the whole texture order in the L2

In order to implement LRU_RO we need to reverse the order of texture accesses in every alternate iteration. Here, the iteration corresponds to a frame. Quads belonging to distinct screen locations access textures, as explained in Section II. The Graphics Pipeline requires primitives to be processed in program order but tiles have no data dependencies and can be processed in any order. Thus the degree of freedom allowed for the change of texture order in a frame has the granularity of a tile. Having said that, note that it is not the aggregated trace to the whole cache but the trace of accesses to each set in the cache that needs to be reversed. This is because the optimality was proved for a fully associative cache and thus applies to each individual set of the cache separately. Figure 8 plots the distribution of the number of texture accesses to a set in the L2, per tile. Note that we do not include zero accesses to this distribution as we observe that in every tile a significant number of sets do not receive any access in the L2. The figure shows us that there is a negligible number of times when a set receives more than one texture access within a tile for ten out of twelve applications. This clearly indicates that if we reverse just the tile order in consecutive frames, we would be closely approximating the full texture access stream reversal for every set, for these ten applications. As for the two other applications, we see that most of the time, the number of accesses is a maximum of two per set in every tile. This implies that for these applications, reversing the tile order will have a texture trace that will stray a bit from the pure reverse texture trace. This observation is supported by the empirical data in Figure 5 as discussed in the previous subsection.

F. Hardware Overhead

In the baseline (LRU_FO), for a given frame, while processing the n^{th} tile, Z-order just requires bit-swizzling of the binary value of n. The Tile Fetcher increments the value of n whenever it starts a new tile, in order to calculate the tile ID of the next tile to be processed in the Z-order. In the baseline, the value of N is reset to 0 at the start of a new frame to repeat the tile order. For alternate reverse order, you would retain the value of n from the last frame and either increment or decrement n, opposite to whatever was chosen in the previous frame. For example, if the first frame starts with $n = 0$, it will keep incrementing and end at $n = T - 1$, where T is the number of tiles in a frame. So now, the second frame will start with $n = T-1$, decrement and then end with $n = 0$. All the operations for a tile, starting from accessing its primitives from the Parameter Buffer, to finally flushing its color buffer to the main memory, requires address calculation with this calculated tile ID. Thus, it is clear to see that reversing the tile order results in no additional cost in hardware.

G. Putting things together

Texture traces can be approximated to circular traces for the properties used in this paper. We propose reversing the order of every other iteration (frame) to have a shorter inter-frame reuse distance for most accesses. To achieve this, we propose traversing tiles in every consecutive frame in the reverse order w.r.t. the previous frame. We proved the quasi-optimality of this proposal in the subsections above and support it with empirical data in Section V.

IV. Evaluation Framework

A. GPU Simulation Framework

We use the TEAPOT [5] simulation infrastructure to evaluate our proposal. TEAPOT is a cycle-accurate GPU simulation framework that allows to run unmodified Android applications and evaluates the performance and energy consumption of the modeled GPU. In order to do that, TEAPOT includes timing and power models based on well-known tools: McPAT [33] for power estimation, and DRAMSim2 [43] for modeling DRAM

TABLE I
EVALUATED BENCHMARKS FROM THE GOOGLE PLAY STORE.

Benchmark	**Alias**	**Installs (Millions)**	**Genre**	**Type**	**Texture Footprint (in MiB)**	**Tex in mainmem accesses (%)**
Crazy Snowboard	CrS	5	Sports	3D	0.7	15.1
Gravitytetris	GrT	5	Puzzle	3D	0.7	3.9
Angry Birds 2	AgB	100	Puzzle	3D	0.9	20.2
Captain America	CAm	5	Action	3D	1.3	33
Derby Destruction Simulator	DDS	10	Racing	3D	1.4	18.2
Sniper 3D	Snp	500	Shooter	3D	1.8	33
Clash of Clans	CoC	500	Strategy	3D	2.0	46.3
Candy Crush Saga	CCS	1000	Puzzle	2D	2.4	52.9
3D Maze 2: Diamonds & Ghosts	Mze	10	Arcade	3D	2.4	34.7
City Racing 3D	CRa	50	Racing	3D	2.6	42.6
Real Steel World Robot Boxing	RSt	50	Strategy	3D	4.2	57.7
Rise of Kingdoms: Lost Crusade	RoK	10	Strategy	2D	6.9	74.5

TABLE II
GPU SIMULATION PARAMETERS.

Global Parameters	
Tech Specs	600MHz, 1V, 32nm
Screen Resolution	1960x768
Tile Size	32x32
Tile Traversal Order	Z-order
Main Memory	
Latency	50-100 cycles
Size	1GiB
Caches	
Vertex Cache	64-bytes/line, 64KiB, 4-way, 1 cycle
Texture Caches (4x)	64-bytes/line, 64KiB, 4-way, 1 cycle
Tile Cache	64-bytes/line, 64KiB, 4-way, 1 cycle
L2 Cache	64-bytes/line, 1MiB, 8-way, 18 cycles

and the memory controllers. Table II shows the parameters employed in our simulations, which resemble those of a contemporary mobile GPU.

B. Benchmarks

We use popular commercial animated graphics applications (games) as benchmarks. We have selected them based on their popularity, in the number of downloads in the Google Play Store, and their variety to cover different types of games.

Table I shows the twelve Android games used to evaluate our technique. We have 2D games like CCS and 3D games like CRa. Games like RoK have a texture footprint of around 6.9MiB in memory whereas CrS and GrT have around 0.7MiB. This affects the benefits of our proposal w.r.t. the baseline (LRU_FO). Textures in RoK contribute to 74.5% of the total accesses to main memory whereas in GrT it contributes to just 3.9%. This affects the caching efficiency for textures translating to a decrease in main memory accesses. Then there are games that are memory-bound and others that are compute-bound, determining how our proposal translates into speedup and energy efficiency.

V. EXPERIMENTAL RESULTS

In this section, we first empirically confirm the quasi-optimality of LRU_RO in simulations, using an ideal architecture where the L2 receives only texture accesses. We also comment on how the LRU is a good candidate to exploit inter-frame reuse distances while holding on to the caching of intra-frame reuses. Secondary to the proof of quasi-optimality, we evaluate the full system and present the performance and energy benefits of LRU_RO with a practical architecture with a shared L2 Cache that receives all types of accesses. In the end, we analyze MRU_FO to emphasize that choosing a replacement policy needs to be done with all types of reuses of the texture in mind along with geometry and the Parameter Buffer accesses. All the above analysis is w.r.t. the baseline, LRU_FO.

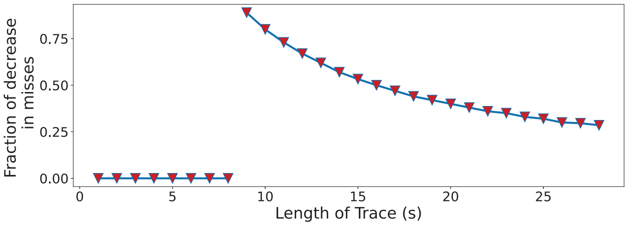

Fig. 9. Decrease in cache misses of LRU_RO w.r.t. LRU_FO, with increasing trace size for a fully-associative cache of size 8.

A. Quasi-Optimality

In Section III, we proved the quasi-optimality of LRU_RO for the texture access stream to the L2. To demonstrate the quasi-optimality for our benchmark suite, we model an experiment with simulations where the GPU has perfect L1 caches (all accesses are hits) except for L1 texture caches. This ensures that no Geometry and Parameter Buffer accesses go to the L2 and the L2 is populated solely by texture accesses. We first set an upper bound for the benefits achieved by LRU_RO w.r.t. the baseline. For a perfect loop trace in a fully associative cache, we know that for $s <= j$, both LRU_RO and LRU_FO produce zero misses in each loop iteration (in the steady state). For $s > j$, LRU_FO produces s misses and LRU_RO produces $s - j$ misses in each loop iteration (in the steady state). Thus the equation for the 'Fraction of Decrease

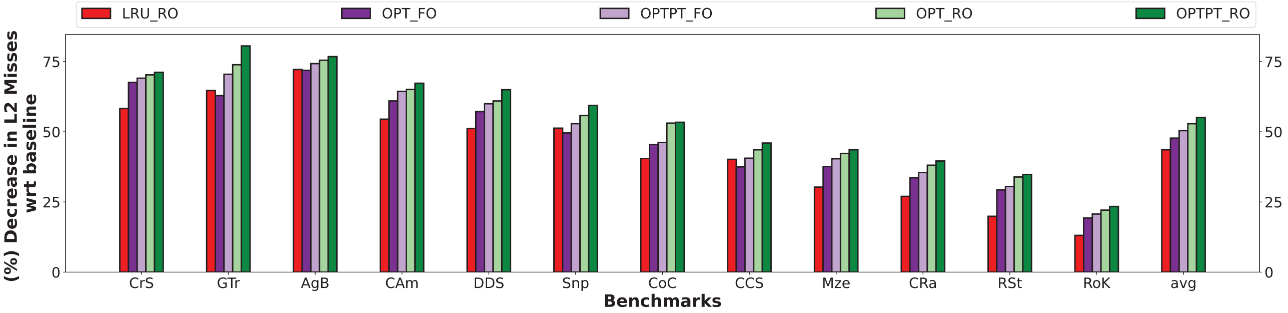

Fig. 10. Decrease in L2 texture misses with different replacement policies, bypassing and tile orders.

in Misses' (FDM) of LRU_RO w.r.t. LRU_FO, for perfect loop traces is as follows.

$$FDM = \frac{s-(s-j)}{s} = \frac{j}{s} \quad \forall \ s > j$$

$$FDM = 0 \quad \forall \ s <= j$$

The equations above tell us that the maximum achievable decrease is $\frac{j}{j+1}$ and it occurs at $s = j + 1$, which, for an associativity of 8, is 0.88 at $s = 9$. The equation also tells us that the decrease is 0 up until $s = j$ and then at $s = j + 1$ achieves the maximum, thereafter behaving like a hyperbola, that decreases asymptotically to zero. Figure 9 shows us this behavior of LRU_RO w.r.t. the baseline for a fully associative cache of size 8. Thus this graph shows the upper bound FDM (for each set in the cache) of our texture traces, that have been approximated to a circular trace.

Now, we showcase the near-optimality of LRU_RO. Figure 10 shows the percentage decrease in L2 misses of the L2 cache w.r.t. the baseline, for LRU_RO and four other theoretical techniques. Note that the applications are arranged in the ascending order of their texture footprint per frame as listed in Table I. The theoretical techniques have two of each order: the forward order and the "alternate reverse order". Both orders have simulations with just OPT and OPT with perfect bypassing (a.k.a. OPTPT). These policies are not realizable in hardware and have been simulated using a stack algorithm.

The first thing we note is the clear relation between the texture footprint and the decrease of L2 misses with optimal caching. This is in accordance with the prediction in Figure 9. We see that on average we get a reduction of 43.6%, 47.7%, 50.4%, 52.9% and 55% in LRU_RO, OPT_FO, $OPTPT\_FO$, OPT_RO and $OPTPT\_RO$, respectively. Table I shows us that CrS, GTr and AgB have a texture footprint lower than 1MiB (the size of the L2 cache) and yet they show a high decrease in L2 misses. This implies a set-mapping load imbalance in texture accesses for these applications and that some sets do get access traces larger than the associativity of the cache, even though the footprint is smaller than the size of the cache. Note that even though benchmarks with a high texture footprint per frame show less decrease in L2 misses, the main point is that LRU_RO reaches very close to the optimal that is simulated with the theoretical techniques. This is indeed the main highlight of this paper, that even if for some benchmarks, the improvement in caching is not as high as the others, it is very close to the maximum achievable caching for textures. Another interesting observation is that, in GTr, AgB, Snp and CCS, the LRU_RO is performing better than OPT_FO as expected from the explanation in section III-B.

We clearly see that LRU_RO is quite close to the optimal policies as the theory indicated before. Even with some dormant access patterns that make the texture access pattern different from circular traces and an approximate reversal (in tile granularity) of the texture order, LRU_RO is able to closely reach the optimal.

These numbers indicate a significant reduction in main memory accesses and thus possibly a reduction in energy. But measuring full system benefits on this ideal state of the system where the L1 Vertex Cache and the L1 Tile Cache are *perfect*, is not meaningful. So, we show power and performance numbers in the next section.

B. Full System Evaluation

Here, we evaluate the performance and energy benefits of LRU_RO w.r.t. the baseline by simulating a GPU with the parameters given in Table II, for our benchmark suite. The benefits of LRU_RO on a shared L2 cache will be proportional to the percentage of texture misses in the L2. Figure 11 shows the distribution of L2 misses arranged in the ascending order of the percentage of texture misses. To better analyze patterns in the rest of this subsection, we plot the benchmarks in this ascending order.

1) Main Memory Accesses: Figure 12 plots the percentage decrease of main memory accesses in LRU_RO. On average, we get a 5.6% decrease and for some applications like AgB and CCS, it goes up to 12.9% and 16.6%, respectively. Note that the percentage for textures in the distribution of L2 misses, as shown in Figure 11, correlates with the decrease in main memory accesses, as expected. The clear outliers to this observation are RSt and RoK. Table I shows that both these applications have the highest texture footprints that are around 4× and 7× the size of the cache. In these cases, even the OPTPT cannot bring a significant decrease in texture accesses, as demonstrated in Figure 10.

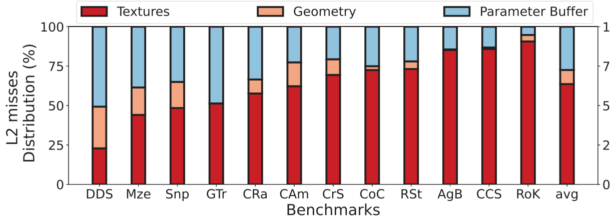

Fig. 11. Distribution of types of L2 misses.

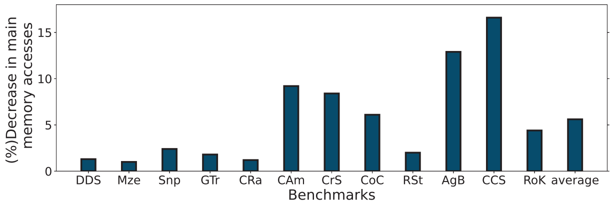

Fig. 12. Decrease in main memory accesses of LRU_RO w.r.t. LRU_FO.

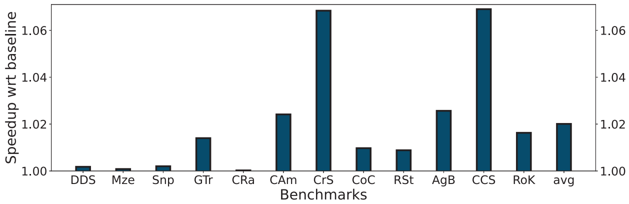

Fig. 13. Speedup of LRU_RO w.r.t. LRU_FO.

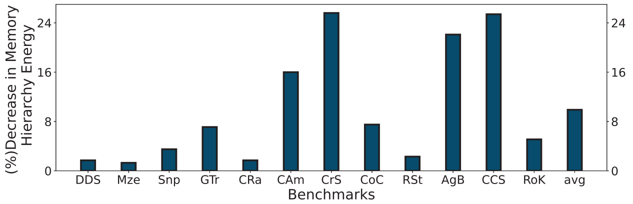

Fig. 14. Decrease in memory hierarchy energy of LRU_RO w.r.t. LRU_FO.

2) Performance: Figure 13 shows the speedup of LRU_RO w.r.t. the baseline. GPUs are designed to hide memory latency. Thus, the translation of improved caching on performance is usually poor. For the evaluated benchmarks, we see an average speedup of 2%, going up to about 7% for CCS and CrS. Performance is crucial for real-time rendering, therefore, this is a noticeable improvement given the fact that it comes at zero cost.

3) Energy: Figure 14 shows the percentage decrease in memory hierarchy energy. We see that Figures 12 and 14 have a close correlation for most benchmarks. CrS is an outlier for this observation because it gets an extra decrease in energy because of the significant decrease in execution time, as can be seen in Figure 13. We see a 9.9% decrease on average and around 24% in CrS, AgB and CCS. It must be noted that decrease in total GPU energy also shows a similar correlation with Figure 12 except for AgB which indicates that this application is not memory-bound w.r.t. energy, for the full GPU. We see a 2.6% decrease in total GPU energy on average. For CCS this goes up to 9.9%. We reiterate that energy efficiency is crucial for mobile GPUs and this decrease has been achieved with zero cost.

C. Other Replacement Policies

The goal of this subsection is to show that choosing another replacement policy to compare with LRU_RO is not straightforward. As explained in section III-C, MRU_FO has the potential to result in a better miss ratio than LRU_FO for loop traces. Figure 15 shows the decrease in L2 misses of LRU_RO and MRU_FO w.r.t. the baseline (LRU_FO) in an L2 that is populated only by textures. We see that, on average, MRU_FO behaves worse than the baseline by 3.6%. In GTr, MRU_FO produces around 2.6× more misses than the baseline (LRU_FO). This shows that MRU might be good to cache inter-frame reuses but destroys the caching of intra-frame reuses which have short reuse distances. This indicates that choosing a replacement policy that safeguards all types of reuses is important to reach optimal caching for texture traces.

VI. Related Work

Works that have addressed the issue of locality in the caches of GPUs, take various directions to propose solutions to exploit it. Specifically for textures in graphics workloads, there is previous work [4] on prefetching texture memory in the L1 texture caches. Another work [8] proposes a NUCA organization for the L1 texture caches to increase their effective overall capacity. Focusing on texture locality-aware workload scheduling to different shader cores with software modifications, Ukarande et al [48] report a 4% speedup when exploiting Texture Cache locality on high-end desktop graphics workloads. Another work [21] also exploits Texture Cache locality by scheduling quads that are closer in screen coordinates. Another work [39] shows improved cache locality in simulations for virtual reality stereo rendering by mapping tiles for left and right eyes to the same shader core.

Other recent works with graphics workloads have explored memory bandwidth reduction in TBR architectures using various methods. Early Visibility Resolution (EVR) [1] is an HSR technique that speculatively predicts the visibility of objects in a scene before the Raster Pipeline to avoid computation and texture accesses of fragments that will eventually be discarded. Rendering Elimination [2] is a technique that detects tiles that produce the same color across adjacent frames to avoid redundant computation and texture accesses. Another work, TCOR [22], explores memory bandwidth reduction by targeting another major source of main memory accesses in TBR architectures, which is the Parameter Buffer. D.Voorhies [49] proposed to rasterize primitive pixels in a boustrophedonic manner to improve texture locality in localized areas within a frame. This is orthogonal and completely differs from our

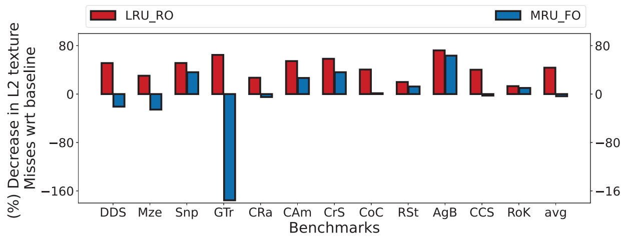

Fig. 15. Decrease in L2 texture misses of MRU_FO and LRU_RO w.r.t. LRU_FO.

full frame reversal, which optimises texture caching for the L2 over multiple frames.

All the above are specifically for graphics and are orthogonal to our proposal and do not address inter-frame cache locality.

As for GPGPU workloads, many more works have targeted cache locality to improve performance. There are many works in literature [9], [24], [29], [31], [32], [47], [50], [56], [57], [59], [61] that explore cache bypassing to improve GPU cache locality. Some works have targeted cache locality across kernel launches for parent-child kernels [52] or generic dependent kernels [16]. Other works like [7], [25], [30] have also studied the impact of warp throttling on locality for GPGPU workloads. Whereas others like [20], [27], [28], [42], [51], [58], have studied the impact of warp-scheduling, within a GPU core. Another work [10] proposes a Cooperative Caching Network (CCN), a ring network among L1 caches of a GPU, to reduce L2 bandwidth demand.

Apart from this, many works like [6], [12]–[15], [17]–[19], [23], [26], [34], [40], [41], [44]–[46], [54], [55], [60] have proposed heuristics to improve caching in caches for general purpose workloads but all of them with hardware overhead. Works like [18], [40] have studied the shortcomings of a circular trace and proposed an effective modification to the LRU in order to improve caching. While this is a solution for general purpose workloads, our solution is specific to TBR GPUs where we do not change the replacement policy but rather the memory access trace itself to reach near-optimal caching.

VII. Conclusions

In this work we have proved that a zero-cost, quasi-optimal L2 Caching mechanism for textures in low-power GPUs is quasi-optimal, by means of a theoretical proof, and then demonstrating it empirically.

We first approximated the texture trace to a generic sequence and formally proved our proposal (Boustrophedonic Frames) for traversing the tiles within a frame in the reverse order of that of the previous frame, to achieve quasi-optimality in the design space of replacement policies, bypassing and tile orders. We then supported the approximation through empirical data extracted from our benchmark suite, composed of real-world animated graphics applications.

Finally, we have demonstrated the quasi-optimality of our proposal through empirical data collected by running our benchmark suite on a simulation framework with a contemporary mobile GPU micro-architecture. Secondary to this, we have shown that our proposal gives a 2% speedup and a 2.6% decrease in total GPU energy. For one benchmark, this goes up to 7% speedup and 9.9% decrease in GPU energy. Note that, since traversing the frame in the reverse order simply requires a switch from an increment operation to a decrement operation at the end of each frame, it does not incur any additional cost.

However, it must be noted that the decrease in texture accesses alone is much higher than the decrease when other types of accesses are introduced in the L2. This shows that there is still potential in trying to leverage this proposal's full potential by using a caching scheme that is orthogonal to our proposal and that targets the other types of data accesses in the L2. TCOR [22] is one such example that targets the Parameter Buffer in the L2 cache and reduces main memory accesses by evicting dead blocks (belonging to the Parameter Buffer) from the L2. This could help enhance the caching for textures in the proposed alternate reverse order.

Acknowledgment

This work has been supported by the CoCoUnit ERC Advanced Grant of the EU's Horizon 2020 program (grant No 833057), the Spanish State Research Agency (MCIN/AEI) under grant PID2020-113172RB-I00, the ICREA Academia program and the AGAUR grant 2020-FISDU-00287. We would also like to thank the anonymous reviewers for their valuable comments.

References

[1] M. Anglada, E. de Lucas, J.-M. Parcerisa, J. L. Aragón, and A. González, "Early Visibility Resolution for Removing Ineffectual Computations in the Graphics Pipeline," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 635–646.

[2] M. Anglada, E. de Lucas, J.-M. Parcerisa, J. L. Aragón, P. Marcuello, and A. González, "Rendering Elimination: Early Discard of Redundant Tiles in the Graphics Pipeline," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 623–634.

[3] I. Antochi, B. Juurlink, S. Vassiliadis, and P. Liuha, "Memory bandwidth requirements of tile-based rendering," *International Workshop on Embedded Computer Systems*, pp. 323–332, 2004.

[4] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Boosting mobile gpu performance with a decoupled access/execute fragment processor," *SIGARCH Computer Architecture News*, vol. 40, pp. 84–93, 2012.

[5] ——, "Teapot: A toolset for evaluating performance, power and image quality on mobile graphics systems," *ACM International Conference on Supercomputing*, p. 37–46, 2013.

[6] M. Chaudhuri, "Pseudo-lifo: The foundation of a new family of replacement policies for last-level caches," *IEEE/ACM International Symposium on Microarchitecture*, pp. 401–412, 2009.

[7] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive cache management for energy-efficient gpu computing," in *International Symposium on Microarchitecture*. IEEE, 2014, pp. 343–355.

[8] D. Corbalán-Navarro, J. L. Aragón, J.-M. Parcerisa, and A. González, "Dtm-nuca: dynamic texture mapping-nuca for energy-efficient graphics rendering," in *2022 30th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2022*. IEEE, 2022, pp. 144–151.

[9] H. Dai, C. Li, H. Zhou, S. Gupta, C. Kartsaklis, and M. Mantor, “A model-driven approach to warp/thread-block level gpu cache bypassing,” in *Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[10] S. Dublish, V. Nagarajan, and N. Topham, “Cooperative caching for gpus,” *Transactions on Architecture and Code Optimization*, 2016.

[11] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, “Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories,” *ACM Siggraph Computer Graphics*, vol. 23, no. 3, pp. 79–88, 1989.

[12] H. Gao and C. Wilkerson, “A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing,” *JWAC 2010 - 1st JILP Worshop on Computer Architecture Competitions: cache replacement Championship*, 2010.

[13] J. Gaur, R. Srinivasan, S. Subramoney, and M. Chaudhuri, “Efficient management of last-level caches in graphics processors for 3d scene rendering workloads,” *Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[14] A. González, C. Aliagas, and M. Valero, “A data cache with multiple caching strategies tuned to different types of locality,” *ACM International Conference on Supercomputing 25th Anniversary Volume*, p. 217–226, 1995.

[15] E. G. Hallnor and S. K. Reinhardt, “A fully associative software-managed cache design,” *IEEE/ACM International Symposium on Computer Architecture*, 2000.

[16] M. Huzaifa, J. Alsop, A. Mahmoud, G. Salvador, M. D. Sinclair, and S. V. Adve, “Inter-kernel reuse-aware thread block scheduling,” *Transactions on Architecture and Code Optimization*, vol. 17, 2020.

[17] A. Jain and C. Lin, “Back to the future: Leveraging belady’s algorithm for improved cache replacement,” *IEEE/ACM International Symposium on Computer Architecture*, pp. 78–89, 2016.

[18] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, “High performance cache replacement using re-reference interval prediction (rrip),” *International Symposium on Computer Architecture*, p. 60–71, 2010.

[19] D. A. Jiménez, “Insertion and promotion for tree-based pseudolru last-level caches,” *IEEE/ACM International Symposium on Microarchitecture*, p. 284–296, 2013.

[20] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, “Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance,” *SIGPLAN*, vol. 48, 2013.

[21] D. Joseph, J. L. Aragón, J.-M. Parcerisa, and A. González, “Dtexl: Decoupled raster pipeline for texture locality,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 213–227.

[22] D. Joseph, J. L. Aragón, J.-M. Parcerisa, and A. González, “TCOR: A Tile Cache with Optimal Replacement,” in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 662–675.

[23] M. Kharbutli and Y. Solihin, “Counter-based cache replacement algorithms,” *International Conference on Computer Design*, pp. 61–68, 2005.

[24] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram, “Access pattern-aware cache management for improving data utilization in gpu,” in *International Symposium on Computer Architecture*, 2017, pp. 307–319.

[25] H.-K. Kuo, T.-K. Yen, B.-C. C. Lai, and J.-Y. Jou, “Cache capacity aware thread scheduling for irregular memory access on many-core gpgpus,” in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2013, pp. 338–343.

[26] A.-C. Lai and B. Falsafi, “Selective, accurate, and timely self-invalidation using last-touch prediction,” *International Symposium on Computer Architecture*, pp. 139–148, 2000.

[27] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, “Cawa: Coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 2015, p. 515–527.

[28] S.-Y. Lee and C.-J. Wu, “Caws: Criticality-aware warp scheduling for gpgpu workloads,” in *International Conference on Parallel Architectures and Compilation*. ACM, 2014, p. 175–186.

[29] ——, “Ctrl-c: Instruction-aware control loop based adaptive cache bypassing for gpus,” in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, 2016, pp. 133–140.

[30] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal, “Locality-aware cta clustering for modern gpus,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, p. 297–311.

[31] A. Li, G.-J. van den Braak, A. Kumar, and H. Corporaal, “Adaptive and transparent cache bypassing for gpus,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015.

[32] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, “Locality-driven dynamic gpu cache bypassing,” in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, p. 67–77.

[33] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures,” *IEEE/ACM International Symposium on Microarchitecture*, p. 469–480, 2009.

[34] H. Liu, M. Ferdman, J. Huh, and D. Burger, “Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency,” *IEEE/ACM International Symposium on Microarchitecture*, pp. 222–233, 2008.

[35] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, “Evaluation techniques for storage hierarchies,” *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[36] S. A. McFarling, “Program analysis and optimization for machines with instruction cache,” Ph.D. dissertation, Stanford University, 1991.

[37] P. Michaud, “Some mathematical facts about optimal cache replacement,” *Transactions on Architecture and Code Optimization 2016*, vol. 13, no. 4, pp. 1–19, 2016.

[38] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, “A sorting classification of parallel rendering,” *IEEE computer graphics and applications*, vol. 14, no. 4, pp. 23–32, 1994.

[39] J.-H. Nah, Y. Lim, S. Ki, and C. Shin, “Z2 traversal order: An interleaving approach for vr stereo rendering on tile-based gpus,” *Computational Visual Media*, vol. 3, no. 4, pp. 349–357, 2017.

[40] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, “Adaptive insertion policies for high performance caching,” *IEEE/ACM International Symposium on Computer Architecture*, p. 381–391, 2007.

[41] J. T. Robinson and M. V. Devarakonda, “Data cache management using frequency-based replacement,” *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1990.

[42] T. G. Rogers, M. O’Connor, and T. M. Aamodt, “Divergence-aware warp scheduling,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, p. 99–110.

[43] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, “Dramsim2: A cycle accurate memory system simulator,” *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[44] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, “The evicted-address filter: A unified mechanism to address both cache pollution and thrashing,” *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 355–366, 2012.

[45] I. Shah, A. Jain, and C. Lin, “Effective mimicry of belady’s min policy,” *2022 IEEE International Symposium on High-Performance Computer Architecture*, pp. 558–572, 2022.

[46] Y. Smaragdakis, S. Kaplan, and P. Wilson, “Eelru: Simple and effective adaptive page replacement,” *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, p. 122–133, 1999.

[47] Y. Tian, S. Puthoor, J. L. Greathouse, B. M. Beckmann, and D. A. Jiménez, “Adaptive gpu cache bypassing,” in *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*. ACM, 2015, p. 25–35.

[48] A. Ukarande, S. Patidar, and R. Rangan, “Locality-aware cta scheduling for gaming applications,” *ACM Trans. Archit. Code Optim.*, vol. 19, 2021.

[49] D. A. Voorhies and N. J. Foskett, “ Method, apparatus and article of manufacture for boustrophedonic rasterization,” in *US Patent 6,650,325*, 2003.

[50] B. Wang, W. Yu, X.-H. Sun, and X. Wang, “Dacache: Memory divergence-aware gpu cache management,” in *International Conference on Supercomputing*, 2015, pp. 89–98.

[51] B. Wang, Y. Zhu, and W. Yu, “Oaws: Memory occlusion aware warp scheduling,” in *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016, pp. 45–55.

[52] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, “Laperm: Locality aware scheduler for dynamic parallelism on gpus,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 583–595.

[53] Wikipedia, “Tiled rendering,” https://en.wikipedia.org/wiki/Tiled_rendering.

[54] W. A. Wong and J. Baer, “Modified lru policies for improving second-level cache behavior,” *International Symposium on High-Performance Computer Architecture*, pp. 49–60, 2000.

[55] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, and J. Emer, “Ship: Signature-based hit predictor for high performance caching,” *IEEE/ACM International Symposium on Microarchitecture*, p. 430–441, 2011.

[56] X. Xie, Y. Liang, G. Sun, and D. Chen, “An efficient compiler framework for cache bypassing on gpus,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 516–523.

[57] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, “Coordinated static and dynamic cache bypassing for gpus,” in *International Symposium on High Performance Computer Architecture*, 2015, pp. 76–88.

[58] Y. Zhang, Z. Xing, C. Liu, C. Tang, and Q. Wang, “Locality based warp scheduling in gpgpus,” *Future Generation Computer Systems*, vol. 82, pp. 520–527, 2018.

[59] C. Zhao, F. Wang, Z. Lin, H. Zhou, and N. Zheng, “Selectively gpu cache bypassing for un-coalesced loads,” in *International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2016, pp. 908–915.

[60] Zhenlin Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems, “Using the compiler to improve cache replacement decisions,” *International Conference on Parallel Architectures and Compilation Techniques*, pp. 199–208, 2002.

[61] X. Zhu, R. Wernsman, and J. Zambreno, “Improving first level cache efficiency for gpus using dynamic line protection,” in *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 2018.