TCOR: A Tile Cache with Optimal Replacement

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

***Abstract*—Cache Replacement Policies are known to have an important impact on hit rates. The OPT replacement policy [27] has been formally proven as optimal for minimizing misses. Due to its need to look far ahead for future memory accesses, it is often reduced to a yardstick for measuring the efficacy of other practical caches. In this paper, we bring the OPT to life, in architectures for mobile GPUs, for which energy efficiency is of great consequence. We also mold other factors in the memory hierarchy to enhance its impact. The end results are a 13.8% decrease in the memory hierarchy energy consumption and an increased throughput in the Tiling Engine. We also observe a 5.5% decrease in the total GPU energy and a 3.7% increase in frames per second (FPS).**

***Keywords*-GPU; Caches; Cache Replacement; OPT; Belady; Energy-efficient;**

I. Introduction

Caches have long proven to be an effective technique in bridging the performance gap between memory and processors. The efficiency of caches is mainly influenced by their size, associativity and replacement policy. Since the advent of caches, a plethora of replacement policies have been explored and evaluated. In 1966, Belady introduced the MIN algorithm [7] that he later proved to be optimum with regard to minimizing the number of misses. In 1970, Mattson et al. proposed and proved optimum the OPT algorithm [27]. These are the two well known, and often wrongly interchanged, optimal replacement algorithms as explained in [28].

Owing to their need to be cognizant of future accesses, these algorithms are normally deemed infeasible. Regardless, these algorithms have been a source of inspiration for many of the state of the art replacement policies and yet each has remained a conceptual tool that is used for evaluating other practical caches.

In this paper we bring the OPT algorithm to the real world for Tile-Based Rendering (TBR) architectures, the predominant architecture for mobile GPUs. We exploit a particular characteristic of this architecture in order to realistically capture and store relevant information of the trace of memory accesses before they are actualized.

In Raster Graphics Systems, GPUs generally contain two distinct parts: a Geometry Pipeline that transforms the geometry of a scene and a Raster Pipeline that paints the transformed scene onto a screen. TBR architectures (further detailed in Section II) introduce a sorting phase between these two parts that bins subsets of this geometry into disjoint segments of the screen called *tiles*, and later retrieves this binned geometry, belonging to each tile, tile by tile to be processed by the Raster Pipeline. This process is called *tiling* and the sorter is called the *Tiling Engine* (see Figure 2). A data structure called the *Parameter Buffer* is stored in memory in order to bin and retrieve this geometry.

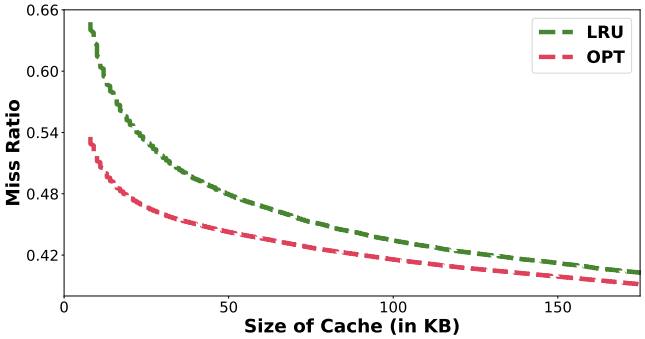

Figure 1: LRU and OPT misses in a fully associative L1 for our mobile graphics benchmark suite.

On an initial analysis on the accesses to the Parameter Buffer for the benchmark suite explained later (see Table II), we plot the behavior of the LRU and the OPT on a fully associative L1 cache. Figure 1 depicts the L1 miss ratios for both replacement policies for an increasing cache size. It shows that OPT causes a drop in miss ratio much faster than LRU. This LRU-OPT gap was seen to be larger for set-associativities of 4 and 8, as shown later in Figure 12, and hence our motivation to implement OPT for mobile GPUs.

Our work strives to hone the memory hierarchy efficiency for the Parameter Buffer. Our primary focus is on reducing the number of misses in all levels of memory. To the best of our knowledge, this is the closest realistic implementation of the OPT replacement algorithm presented in literature. While this is widely known to be impractical for most scenarios, TBR architectures are unique in that they build and use up the Parameter Buffer in consecutive pipeline stages, and that the stage that builds the data also possesses the information to infer future reads. In this work, we propose a practical approach to derive this information about future accesses during binning, store it and then use it to drive the replacement policy of the Tile Cache.

We also propose an alternative layout for the Parameter Buffer to improve the load-balancing on the sets of the

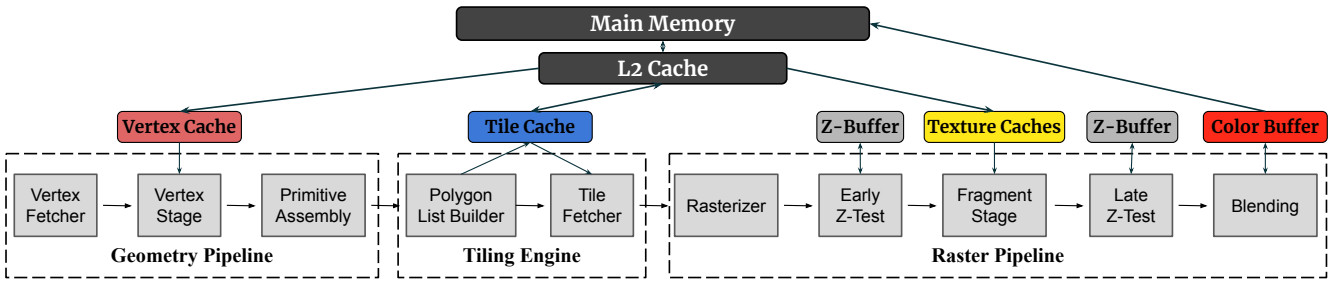

Figure 2: The Graphics Pipeline of a TBR GPU.

caches and thus further reduce conflict misses. In addition, we provide a way to reduce the write-back operations of the Parameter Buffer into the main memory by identifying dead blocks of data and prioritizing the retention of potentially live blocks of data.

We refer to this novel memory hierarchy organization as Tile Cache with Optimal Replacement (TCOR), and we show that it provides a 13.8% energy reduction in the memory hierarchy and a 5x speedup in the Tiling Engine on average, for a set of representative animated graphics applications. We also report a decrease of 5.5% in total GPU energy and a 3.7% increase in FPS, on average.

To summarize, this paper makes the following key contributions:

- Presents an implementation of the OPT replacement algorithm, for the first time in literature, for the Tile Cache of mobile GPUs.
- Presents a new replacement policy in the L2 to amplify the effects of OPT on higher levels of the memory hierarchy.
- Shows that this new Tile Cache architecture provides an energy reduction of 13.8% in the memory hierarchy, on average, for a set of real world animated graphics applications.

The rest of this paper is organized as follows. Section 2 presents some background on GPUs, with special emphasis on how they manage the Parameter Buffer. Section 3 describes the proposed TCOR cache organization. We describe the tools and workloads used to evaluate our technique in Section 4. In Section 5, we present our experimental results and analysis. In Section 6, we review some related work and Section 7 summarizes the main conclusions of the paper.

II. Background

Mobile GPUs typically implement a Tile-Based Rendering (TBR) architecture. The idea for TBR architectures was initially proposed to facilitate parallel rendering [9], [29]. *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. [4], a TBR architecture reduces the total amount of external data traffic by a factor of 1.96 compared to a traditional GPU architecture.

A. 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* that are 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, the geometry needs to be sorted into subsets that will individually be able to fully render the image for each of these tiles. There are various methods of sorting. In the sorting classification of rendering techniques, as described in [29], TBR can be classified as a Sort-Middle technique. 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 and the Vertex Stage starts fetching vertices from memory using an L1 Vertex Cache. It then transforms them according to a vertex program provided by the user. The Primitive Assembler takes the vertices in program order 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 for each tile with all the primitives that overlap it. This

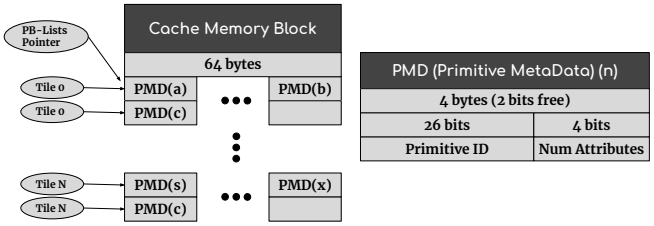

Figure 3: PB-Lists section of the baseline Parameter Buffer.

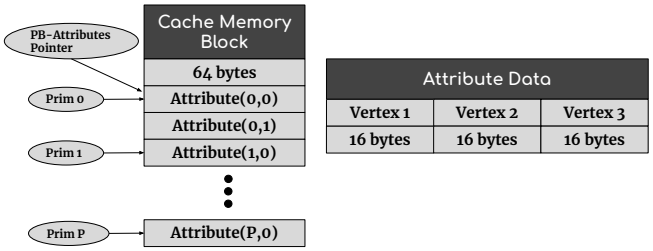

Figure 4: PB-Attributes section of the baseline Parameter Buffer.

data is known as the *Parameter Buffer*. For this purpose, the Polygon List Builder takes each primitive generated by the Primitive Assembly in program order and appends it to each list of every tile it overlaps. For each primitive, the Raster Pipeline needs to know its *attributes*, which describe characteristics of its vertices (e.g., color, normals, texture coordinates, etc.). Since attributes occupy significant space and primitives may overlap many tiles, the attributes of each primitive are stored only once in the Parameter Buffer, and the per-tile lists contain only the IDs of the primitives.

The Parameter Buffer is built and used up in the same frame. 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 its primitives are put into a FIFO queue for the Raster Pipeline to consume.

The Raster Pipeline renders each tile sequentially. For this purpose, it takes each primitive from the FIFO queue and identifies which pixels of the corresponding tile are overlapped by the primitive. It then uses interpolation to calculate attributes for each pixel and then generates quantas of data called quads that consist of interpolated attributes for a group of four pixels. These quads are put into a FIFO queue from where the Early Z-Test takes them up to eliminate quads that lie behind another opaque quad that was previously processed. This stage uses a buffer called the *Z-Buffer* to store the minimum depth of previously processed quads. The remaining quads are put in a FIFO queue from where the Fragment Stage takes them and assigns them to a Fragment Processor. The Fragment Processors compute an initial color for each pixel of a quad, taking into account the lighting and textures provided by the program. After these quads are processed, the output color of these quads are put into another FIFO queue from where the Blending Unit takes them up and computes the final color of pixels depending on the transparency of each quad. These final colors are stored in the Color Buffer. Some rendering techniques require changing the depth of fragments in the Fragment Stage, in which case the Early Z-Test is disabled and the Late Z-Test 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. The Color Buffer is flushed to the Frame Buffer in main memory after a tile has been completely processed.

B. Parameter Buffer

As introduced before, the Parameter Buffer stores the information needed for the Raster Pipeline to render each tile independently. For each tile, the Rasterizer needs the vertices' attributes of the primitives that overlap that tile. This information is stored in two sections: PB-Lists and PB-Attributes.

Figure 3 illustrates the layout of PB-Lists in the baseline design. PB-Lists stores a list for each tile containing the primitives that overlap it. Since a primitive may appear in multiple lists, only primitive IDs are stored in these lists, whereas the attributes of each primitive are stored in PB-Attributes. Each primitive of a list has its corresponding metadata or PMD (Primitive Meta Data). A PMD in the baseline stores a *Primitive ID* and the *Number of attributes* that the primitive has. For convenience, the address of the first attribute of a primitive is used as the Primitive ID. This attribute is stored in PB-Attributes followed by the rest of the attributes for a primitive. Sixteen such PMDs are stored in a block of memory (we assume a cache line of 64 bytes). As each tile is allotted a maximum of 1024 primitives, the list for the next tile begins 64 blocks after the current one. The first primitive list begins at a predetermined address in memory that we refer to as PB-Lists Pointer.

Figure 4 illustrates the layout of PB-Attributes in the baseline design. PB-Attributes stores the primitives in the order that they arrive in the Polygon List Builder. Each primitive has a variable number of attributes and each attribute occupies 48 bytes (16 bytes for each vertex of the primitive) and is block aligned. The first attribute of the first primitive begins at a predetermined address in memory that we refer to as PB-Attributes Pointer.

C. Tile Cache

The Tile Cache is used by the Tiling Engine to access the Parameter Buffer. The Polygon List Builder builds the Parameter Buffer and writes it into memory and the Tile Fetcher reads it later from memory, both using the Tile Cache. When a primitive is binned, a write request to PB-Lists is generated to write its PMD for each tile it overlaps. Then, a number of write requests to PB-Attributes are generated in order to store the attributes of that primitive.

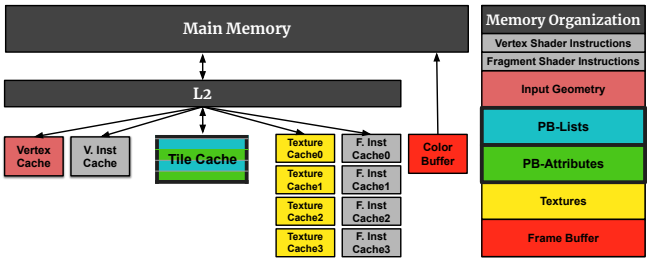

Figure 5: Baseline memory hierarchy and memory organization.

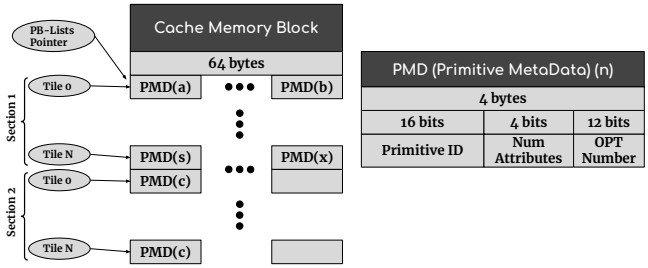

Figure 6: New layout of the PB-Lists section of the Parameter Buffer.

For each tile, the Tile Fetcher generates a read request to that tile's list in PB-Lists. The PMDs obtained help generate read requests for the attributes of all primitives in that tile. Although memory requests may be completed out of order, the Tile Fetcher has to pass the primitives to the Rasterizer in the original order. For this purpose, when the Tile Fetcher receives a reply from the memory hierarchy, the data is stored in the cache, but primitive requests are tracked in a temporary FIFO queue and a primitive is not put in the output queue until it is completed and is the oldest one in this temporary queue.

D. Memory Organization

Figure 5 depicts the main memory regions of a graphics application (on the right) and the memory structures used to store and access them (on the left). As it can be seen, there are multiple L1 caches for instructions and data, connected to a shared L2 cache, which is backed up by main memory.

III. TCOR

Our main goal in this work is to implement a Tile Cache with an Optimal Replacement (TCOR) policy. As illustrated in Figure 1, an optimal replacement policy, OPT, provides significant benefits over a typical LRU scheme.

Conceptually, OPT requires a look into future memory accesses in order to make a decision on replacement. In particular, upon a cache miss, out of all the candidate lines in the cache at that time, it replaces the one that will be accessed the farthest away in the future. Candidate lines are all the lines that are currently present in the cache for a fully-associative cache, or in the set where this request will be placed for a set-associative cache. Thus, the cache needs to be cognizant of future accesses to be able to determine a replacement. The main challenges associated with the implementation of OPT are listed below.

1) A trace of future memory accesses needs to be procured to use OPT in a timely fashion.
2) Such a trace is expected to be large and it needs to be stored and retrieved with ease.
3) Information relevant to making a replacement must be present in the cache during replacement.

A. Our Solution

We obtain the trace of future memory accesses from temporary data produced during the binning of primitives in the Polygon List Builder. While binning a primitive, all tiles that are overlapped by this primitive are identified. Note that each primitive can appear in a given list (i.e., in a particular tile) at most once, and the order in which the Tile Fetcher will process the tiles is fixed and known beforehand. When the Polygon List Builder generates the Parameter Buffer, every time a new primitive is binned, for each list in PB-Lists where it is appended, the ID of the next tile that will use this primitive is included in the primitive metadata (PMD), in addition to the primitive ID and the number of attributes (see Figure 6). We dub this field the OPT Number.

Note that when the Parameter Buffer is built by the Polygon List Builder, it does not perform any read but it only writes each primitive exactly once into PB-Attributes. All these writes will be compulsory misses that the replacement policy cannot affect. Thus the accesses (writes) by the Polygon List Builder need not use OPT.

Whenever a primitive is later read by the Tile Fetcher, an access is generated for its PMD in the PB-Lists. The OPT Number for that primitive in that particular tile is retrieved from its PMD and stored in cache with the line assigned to that primitive. Thus at the time of replacement, each line will have information about its next access. OPT will select the line whose next access is the farthest away, out of the lines in the same set.

B. A New Layout for the Parameter Buffer

We introduce a new layout for PB-Lists as shown in Figure 6. In the baseline, PB-Lists stores the list of primitives consecutively in memory. Our benchmarks indicate to us that most of the space dedicated to each list in memory remains empty as the number of overlapped primitives are much lesser than the maximum. This sparsity in PB-Lists gives rise to higher cache conflicts among consecutive tiles since their data is separated by a relatively large power of 2 (1024 primitives, 64 memory blocks). This causes most of the data to be mapped just to a few sets of the cache, which results in many conflict misses.

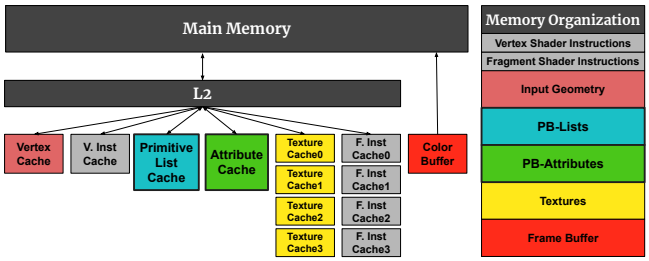

Figure 7: The proposed Tile Cache is split into a Primitive List Cache and an Attribute Cache.

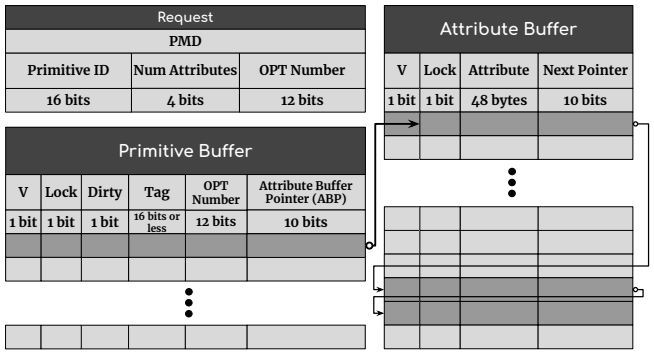

Figure 8: The Attribute Cache.

To combat this, we introduce a new layout for PB-Lists as shown in Figure 6. Here, the lists of primitives are stored in an interleaved manner. The first tile begins at PB-Lists Pointer in memory and stores a maximum of 16 PMDs in that block. The list for the second tile begins at the next block and so on. This collection of part of the lists of all tiles (one memory block per tile) is marked as a section in the figure. The next 16 PMDs of each list is stored after this first section and so on.

C. The New Tile Cache

The main goal of the Tile Cache is to speedup and reduce the energy consumption of the accesses to the Parameter Buffer. Owing to the fact that the PB-Lists and the PB-Attributes sections have very different access patterns, we split the Tile Cache into two caches as shown in Figure 7, each one accessing one of the two sections of the Parameter Buffer. The one accessing PB-Lists is called the Primitive List Cache and the other one the Attribute Cache.

1) The Primitive List Cache: The Primitive List Cache is used to access the PB-Lists section of the Parameter Buffer. This data is written once by the Polygon List Builder, but writes happen at the granularity of primitives, so there is reuse that can be exploited at the memory block level, since each memory block contains 16 PMDs (that fits the size of a cache line, 64 bytes). Later, each memory block is read only once by the Tile Fetcher and the primitives it contains are processed sequentially. Since PB-Lists accesses represent a very minor percentage of the total memory bandwidth (a PMD is four bytes long whereas an average primitive has around 3 attributes, leading to 192 bytes) and its reuse is quite limited, we use a conventional cache with an LRU replacement policy for the Primitive List Cache.

2) The Attribute Cache: The Attribute Cache is dedicated to accessing the PB-Attributes section of the Parameter Buffer. Attributes belonging to a primitive are always accessed together, and are identified by their Primitive ID. Thus, the granularity of access to this cache is a primitive. Each primitive has a variable number of attributes, so we decouple the Attribute Cache into two structures: the Primitive Buffer and the Attribute Buffer, as shown in Figure 8. The Attribute Buffer stores the attributes themselves whereas the Primitive Buffer stores pointers to those attributes.

Each line of the Primitive Buffer corresponds to a primitive and stores three control bits, the tag, the OPT Number and the *Attribute Buffer Pointer (ABP)*. The Primitive Buffer is indexed with the Primitive ID and each line in the cache is tagged with the most significant bits to uniquely identify the stored primitive. Each line also possesses a valid bit, a lock bit and a dirty bit. The lock bit is used to prevent the replacement of a cache line until the primitive that it stores has been processed by the Rasterizer. The data that the line holds (ABP) is actually a pointer to the first attribute of that primitive in the Attribute Buffer. Lastly, the OPT Number in each line is used for performing the OPT replacement.

Note that the addition of the ABP and the OPT Number adds 22(10+12) bits to each primitive in the Attribute Cache. But the baseline stores each primitive's attributes in separate lines and each line stores a tag. Since each primitive has three attributes on average, we save 32(16*2) bits per primitive. Thus, there is no overhead in terms of area.

An XOR-based indexing function [12] is used for mapping each primitive to a set. As explained in [36], this mapping scheme helps in reducing conflict misses.

On the other hand, the Attribute Buffer stores all the attributes corresponding to each primitive present in the Attribute Cache. They are stored as a linked list of entries in the buffer, since their number is variable. Each buffer entry contains an attribute, a valid bit, a lock bit and a pointer to the next entry in the list (null for the last attribute).

A linked list of free entries is also maintained within the buffer to manage the allocation of available entries. A new primitive can be added to the Attribute Cache provided there are enough free slots in the Attribute Buffer to store all of its attributes.The working of the cache is explained next.

3) Reads: Reads to the Tile Cache are done by the Tile Fetcher when a tile is rasterized. In particular, for each tile, the Tile Fetcher reads all its PMDs from PB-Lists (using the Primitive List Cache) and for each PMD, it generates a read request for its attributes to the Attribute Cache. The goal

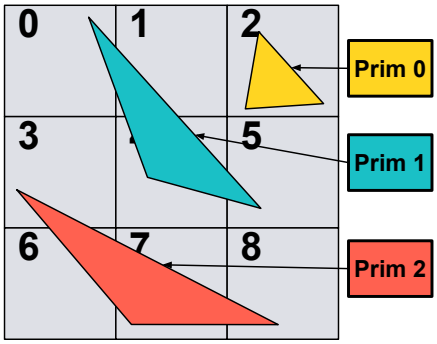

Figure 9: The Example Frame.

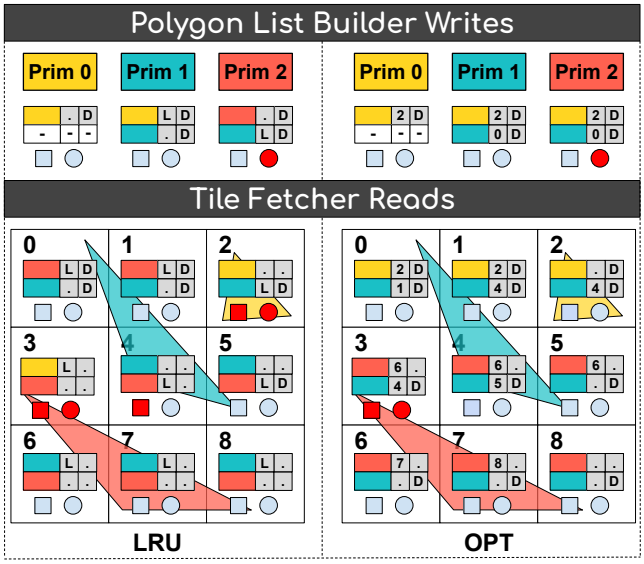

Figure 10: The Example: LRU vs OPT Cache states.

of these reads is to pass the attributes of a primitive to the Rasterizer. The attributes are cached in the Attribute Cache for later reuse, since a primitive may overlap multiple tiles. To pass the attributes to the Rasterizer, instead of copying them, the Attribute Buffer Pointer (ABP) is passed to the Rasterizer in the Tile Fetcher output queue, for the sake of energy efficiency.

A read request has the three fields provided by its PMD (see Figure 8): the ID of the primitive, the number of attributes of this primitive, and the OPT Number used for replacement. Below we detail the main operations to be performed for reads:

Hit - If the requested primitive is found in the Attribute Cache, its corresponding line in the Primitive Buffer and its attributes in the Attribute Buffer are locked until the Rasterizer consumes that primitive (in the Attribute Buffer, it suffices to lock the first attribute, since the rest are linked and will not be released until the first one is). The OPT Number of that line is then updated with the one provided by the request. The Attribute Buffer Pointer (ABP) for that primitive is pushed onto the output queue for the Rasterizer to pick up and access.

Miss - If the Tile Fetcher request results in a miss, the Primitive Buffer evicts a line, reserves it for the current miss and locks it until the request for the current miss is serviced. The cache also makes sure that there are sufficient free slots in the Attribute Buffer for all the attributes in the current miss. In case of a dearth of space, more primitives are evicted using OPT. The cache then generates separate requests for all the attributes of the current miss and puts them in the MSHRs.

When the misses for all the said attributes are serviced, the Tile Fetcher pushes the Attribute Buffer Pointer (ABP) to the output queue.

Rasterizer Read - The Rasterizer obtains from its input queue (i.e., the Tile Fetcher's output queue), the pointer to the first attribute of a primitive in the Attribute Buffer (the ABP). It accesses all the attributes of this primitive from the Attribute Buffer and then it unlocks all these entries in the Attribute Buffer.

4) Writes: Writes to the PB-Attributes are done only by the Polygon List Builder every time a new primitive is processed. Each attribute is written only once, so all writes will result in a miss. The OPT Number of a write request is the ID of the first tile that will access this primitive in the Tile Fetcher.

When the Polygon List Builder writes a new primitive in PB-Attributes, a write request is made to the L1 Attribute Cache. The cache checks if there are free lines in the corresponding set of the Primitive Buffer, and if not, it chooses either to evict a primitive or bypass the current write to the L2. For this purpose, the OPT Number of each line in the corresponding set is compared with the OPT Number of the request to find the line with the highest OPT Number. If the OPT Number of this line is greater than that of the request, it means that the primitive in that line will be accessed by the Tile Fetcher after the primitive in the current request. In this case, the line is evicted and the request is written in L1. If the OPT Number of this line is lesser than that of the request, it means that all the primitives in the set will be accessed by the Tile Fetcher before the primitive in the current request. In this case, the request is bypassed and written into L2. Finally, if the OPT Number of this line is equal to that of the request (they both belong to the same tile), the request is still bypassed and written into L2.

5) Evictions: Evictions are done only to unlocked primitives. Unlocked lines are identified by checking the lock bit associated with the line as well as the lock bit of its first attribute in the Attribute Buffer. Before eviction, the Attribute Cache checks if the line is dirty and if yes, proceeds to generate a write request to the L2 for each of the attributes belonging to that primitive.

6) Replacement Policy: Upon a miss, the replacement policy picks the line with the greatest OPT Number among all unlocked lines of the set where the new primitive is going to be stored. It must be noted that the replacement policy is only applied when there are no empty lines in a set.

7) An Example: As an illustrative example, let us assume that there are 3 primitives and 9 tiles in a frame. Let us also assume that the Tile Fetcher follows a scanline order to process the different tiles, that the cache is fully associative and has space for only two primitives. Figure 9 shows the layout of these primitives in the frame.

The Polygon List Builder will make 3 writes to the Attribute Cache, one for each primitive. The Tile Fetcher will make 9 reads to the Attribute Cache, one for each tile, as each tile is overlapped by exactly one primitive.

Figure 10 compares our implementation of OPT with that of LRU for this example. The figure shows twelve cache states after each access by both Tiling Engine stages. The first three states correspond to the three writes of the Polygon List Builder and the remaining nine correspond to reads of the Tile Fetcher. The initial state (not shown) is when there are no primitives in the cache. The cache state is represented by two rows corresponding to both cache lines. There are three columns in each row. The first one indicates which primitive occupies the line (a white color signifies an unoccupied line). The second column represents the LRU value (the least recently used line is marked as L) and the OPT Number for the LRU and the OPT replacement policies respectively. The last column indicates whether the line is dirty. The state of the cache also indicates an L2 read and an L2 write with a red square and a red circle respectively. We can see that the first L2 write occurs for the third write in both cases, but for LRU it is a write-back on eviction whereas for our OPT it is a bypass. This is because the pink primitive has a OPT Number of 3 which is greater than all the primitives in the current state of the cache. When the yellow primitive is later requested by the Tile Fetcher in Tile 2, it results in an L2 read and write for the LRU since the primitive is not present and one of the present primitives (the pink primitive), which is dirty, needs to be evicted. Since OPT has retained the yellow primitive, it avoids this miss. Both policies have an L2 read and write in Tile 3 but notice that our technique evicts the yellow primitive, which will never be accessed again, whereas the LRU retains it. This results in another L2 read for the LRU in Tile 4 to fetch the blue primitive.

D. TCOR Enhancements for the L2

1) Identifying Dead Cache Lines: As commented above, the Parameter Buffer is built and used up within each frame. Each part of the Parameter Buffer becomes dead (i.e., no further reads will occur) during the course of Tile Fetching, after its last use in each frame. In case of PB-Lists, the list for a given tile becomes dead after that tile has finished being processed by the Tile Fetcher. As for PB-Attributes, all attributes of a given primitive become dead after the last tile that includes that primitive has finished being processed by the Tile Fetcher. The information about the last tile that uses this data can be derived from the Polygon List Builder just like the OPT Number is derived. The idea is to propagate this information for both sections of the Parameter Buffer along with each block of memory and then use that information to identify these blocks as dead in the L2 once the corresponding tiles have finished being processed by the Tile Fetcher.

To facilitate this idea, two fields are added to each line in the L2. Firstly, the replacement policy should be able to identify which cache lines hold data belonging to the Paramater Buffer. A 2-bit field is added to each line in the L2 to indicate whether the data in a line belongs to PB-Lists, to PB-Attributes or to neither of them (L2 is shared by several other L1 caches as shown in Figure 7). The second field is a 12-bit value that stores the ID of the last tile that will use this line in case the data belongs to the Parameter Buffer.

Each line containing data from PB-Lists belongs to a single tile (which is the last to access it). The required Tile ID for these lines can be derived from its block address and the starting address of PB-Lists. As an example, if the initial address of PB-Lists is a relatively large power of two and the number of tiles is also a power of two, then the tile ID can be inferred just by extracting the least significant bits of the memory block address. This is because the Tile Lists are stored in an interleaved manner (see Figure 6). If powers of two are not used, then a subtraction of the block address from the initial address followed by a modulo operation with the number of tiles, will produce the tile ID.

At the same time, a line containing data from PB-Attributes (a single attribute per line) must be explicitly tagged with the ID of the last tile that will access it. This Tile ID is known when the Polygon List Builder computes the OPT numbers. Since each attribute only has 48 bytes and a block in memory is 64 bytes, the Polygon List Builder stores, in these unused bits, the 12 bits of the ID of the last tile that will access this memory block.

The L2 control logic also stores the Tile ID of the last tile that finished processing in the Tile Fetcher. This value is initially set to `NULL`. Every time the Tile Fetcher finishes processing a tile, it sends a signal to the L2 and this value is incremented.

Using the two new fields in the L2 cache lines and the Tile ID of the last tile processed by the Tile Fetcher, we can infer which lines in the L2 are dead. This information is inferred and used every time a line has to be replaced in a given set, as described below.

2) Replacement Policy Modification: Every time a line is to be replaced, we first identify dead lines in the appropriate set. The replacement policy of the L2 is modified to

Table I: 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, 12 cycles

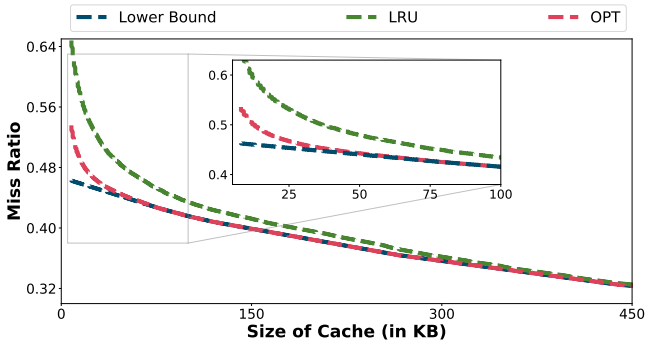

Figure 11: LRU and OPT misses in a fully associative L1 for our Mobile Graphics Benchmark Suite (the inner graph zooms in the marked area).

prioritize the eviction of dead lines, which are obviously the best candidates since they will no longer be used. In case the evicted line is dead, it does not have to be written back to Main Memory even if it is dirty. With this, we reduce write-backs to main memory. The second priority for replacement is given to non Parameter Buffer data whereas the live Parameter Buffer data is given the lowest priority. Within each priority set, LRU is used.

The rationale for this prioritization follows. The L2 not only caches Parameter Buffer data but also textures, vertices and instructions. Texture, vertices and instruction cache lines are always in a clean state whereas Parameter Buffer cache lines may be dirty. Replacing a dirty line is more expensive as it needs to be written back to Main Memory.

IV. Evaluation Framework

A. GPU Simulation Framework

We use the TEAPOT [5] simulation infrastructure to evaluate our proposals. TEAPOT is a cycle-accurate GPU simulation framework that allows to run unmodified Android applications and evaluate 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 [25] for power estimation, and DRAMSim2 [33] for modelling DRAM and the memory controllers. Table I shows the parameters employed in our simulations, which resemble those of a contemporary mobile GPU.

B. Benchmarks

We use popular commercial animated 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 II shows the ten Android games used to evaluate our technique. We have 2D games like CCS and 3D games like CRa. Games like RoK have a Parameter Buffer that leaves a footprint of around 0.2MiB in memory whereas DDS leaves around 1.8MiB. The average number of primitives overlapped per tile in TRu is 11 whereas that in DDS is 20.9 primitives. All these factors affect the effectiveness of our technique. A larger Parameter Buffer means a higher percentage of capacity misses which cannot be ameliorated by cache replacement techniques. Whereas a higher number of tiles overlapped per primitive means higher reuse, which increases the effectiveness of the replacement algorithm. RoK accesses textures with a footprint of around 6.8MiB while SWa has a 0.4MiB footprint. CCS includes shader programs with an average of 4 instructions per pixel whereas DDS' shader programs have an average of 20 instructions. This affects the propagation of the effectiveness of TCOR to the L2. It also affects the efficiency of the proposed L2 eviction policy, since the L2 is shared by all the L1 caches, including instructions, textures, primitives and attributes.

V. Experimental Results

In this section we first explore the isolated benefits of the TCOR optimal replacement policy in the Attribute Cache. Then, we evaluate the effects of TCOR on memory traffic, power and performance.

A. Benefits of OPT Replacement Policy

In the introduction section (see Figure 1) we already showed that, for a fully associative L1 Attribute Cache, OPT provides a significant drop in cache misses for a wide range of cache sizes when compared to LRU. Here, we complement this study by showing that OPT reaches a lower bound on the total cache misses with a much smaller cache capacity than LRU.

We also show that OPT falls to the lower bound faster than LRU for increasing set-associativity. Finally, we contrast OPT, LRU, MRU and a state-of-the-art replacement policy (DRRIP) against the lower bound, for varying cache sizes.

To compute this lower bound on the total number of cache misses, we note that a unique characteristic of the accesses to PB-Attributes is that each memory block will be requested at least twice over the course of the tiling process. Each primitive's attribute is written exactly once

Table II: Evaluated benchmarks from the Google Play Store.

Benchmark	Alias	Installs (Millions)	Genre	Type	Parameter Buffer Footprint (in MiB)	Avg Prim Re-use
Candy Crush Saga	CCS	1000	Puzzle	2D	0.17	5.9
Sonic Dash	SoD	100	Arcade	3D	0.14	6.9
Shoot Strike War Fire	SWa	10	Shooter	3D	0.28	3.7
Temple Run	TRu	500	Arcade	3D	0.55	2.8
City Racing 3D	CRa	50	Racing	3D	0.86	2.0
Rise of Kingdoms: Lost Crusade	RoK	10	Strategy	2D	0.2	3.6
Derby Destruction Simulator	DDS	10	Racing	3D	1.81	1.4
Sniper 3D	Snp	500	Shooter	3D	0.71	1.47
3D Maze 2: Diamonds & Ghosts	Mze	10	Arcade	3D	1.22	2.4
Gravitytetris	GTr	5	Puzzle	3D	0.12	6.9

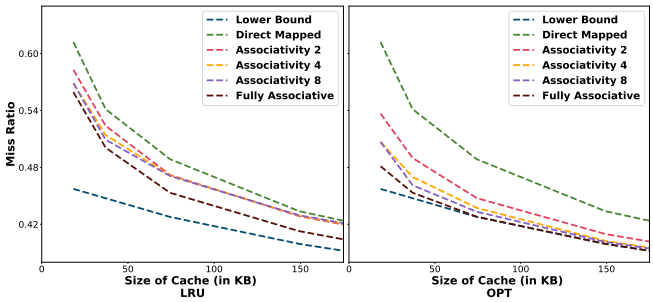

Figure 12: LRU and OPT misses in a set-associative L1 for our Mobile Graphics Benchmark Suite.

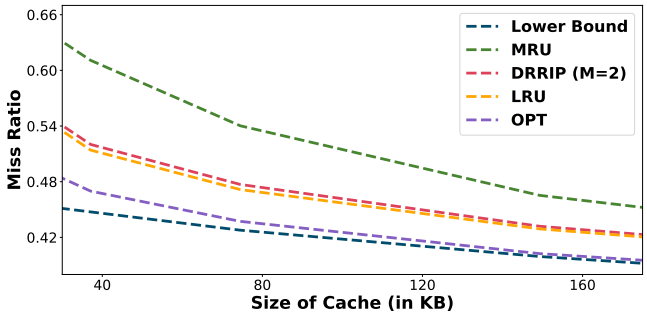

Figure 13: LRU, MRU, DRRIP and OPT miss ratios in a 4-way L1 for our Mobile Graphics Benchmark Suite.

by the Polygon List Builder and read at least once by the Tile Fetcher. Each write will result in a compulsory miss. For reads, the primitives that cannot fit into the Attribute Cache at the end of the Polygon List Builder will generate a miss the first time they are read (and potentially additional capacity misses). For instance, if we have 1000 primitives and the Attribute Cache has a capacity for 128 primitives, there will be at least 872 (1000-128) read misses, since at the end of the Polygon List Builder, 872 primitives will not be present in the Attribute Cache.

In general, let the total number of primitives be TP and the maximum number of primitives in the Attribute Cache be CP. Thus, adding the compulsory write and minimum read misses, we can compute the following lower bound on the total number of misses (LB) for any cache associativity and replacement policy:

$$LB \geq TP + (TP - CP) \qquad \forall\, CP < TP$$
$$LB \geq TP \qquad \forall\, CP \geq TP$$

In Figure 11, we plot this lower bound together with the miss ratios of both the LRU and OPT replacement policies, for a fully associative L1 Attribute Cache and a range of cache sizes. In this figure, we can see that OPT reaches this lower bound near 55KiB whereas LRU reaches it at 375KiB. In other words, OPT can reach its maximum potential for a cache size that is 6.8 times smaller than that using an LRU.

Next, we show the benefits of OPT for set-associative caches. Figure 12 plots the miss ratio for an L1 Attribute Cache for increasing cache sizes and varying set associativity for LRU and OPT. With increasing associativity, the miss ratio falls to the lower bound much earlier for the OPT than that for the LRU. Observe that the plot for an associativity of 2 with OPT shows very similar benefits to those achieved by the fully associative LRU.

Figure 13 compares OPT with LRU, MRU and (DRRIP [22]) in a 4-way cache for a range of cache sizes. MRU has the highest Miss Ratio followed by DRRIP which is then followed closely by LRU. OPT shows a miss ratio that falls quickly to the lower bound, with an increasing cache size. DRRIP has been shown to outperform LRU for CPU workloads (with mixed access patterns) for an LLC cache where the reuse pattern is filtered of temporal locality by lower level caches [22]. In this figure, we see that for our L1 cache, with its unique workload for the Parameter Buffer, DRRIP shows no benefits over LRU.

B. Evaluating TCOR

We now evaluate the benefits of TCOR on a baseline GPU with the configuration described in Table I. We also report results for a larger Tile Cache of 128KiB and an associativity of 4. To match TCOR with the baseline, we assume a 16KiB Primitive List Cache and a 48KiB Attribute Cache and in

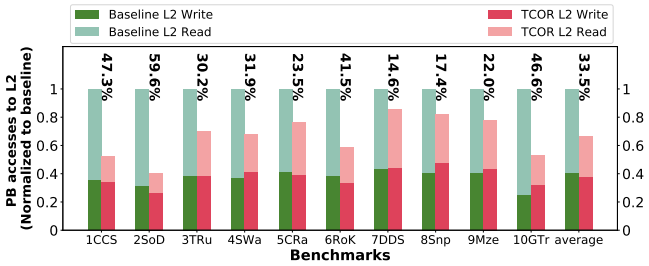

Figure 14: Parameter Buffer accesses to L2 normalized to the baseline (64KiB Tile Cache).

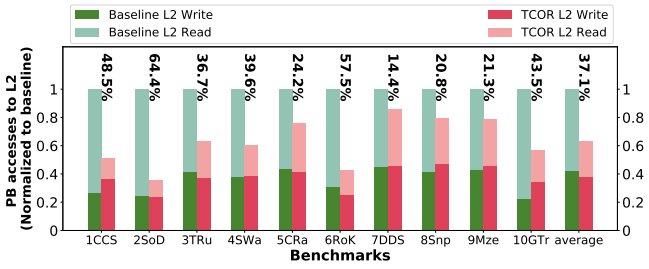

Figure 15: Parameter Buffer accesses to L2, normalized to the baseline (128KiB Tile Cache).

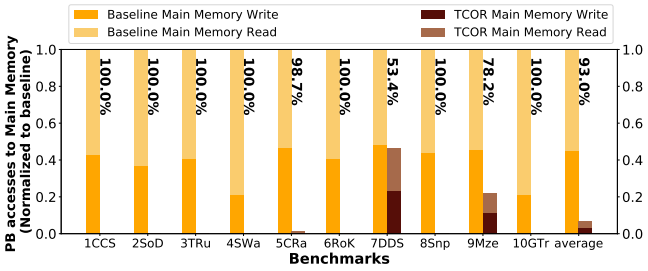

Figure 16: Parameter Buffer accesses to Main Memory normalized to the baseline (64KiB Tile Cache).

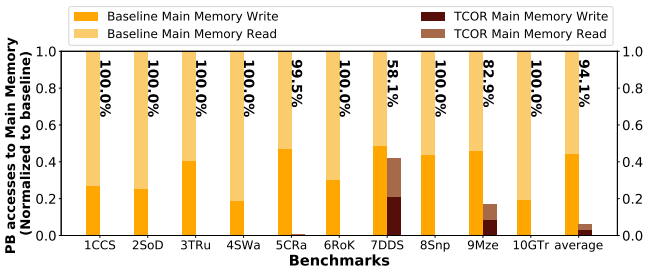

Figure 17: Parameter Buffer accesses to Main Memory normalized to the baseline (128KiB Tile Cache).

the second case, we have a 16KiB Primitive List Cache and a 112KiB Attribute Cache.

1) Memory Traffic Reduction: TCOR is aimed at increasing the effectiveness of caching the Parameter Buffer in the memory hierarchy. In order to assess the effectiveness of these improvements, we measure the number of Parameter Buffer accesses to each level of the memory hierarchy.

L1. There are several differences between the Tile Caches of the baseline and TCOR. Firstly, the baseline has only one L1 dedicated to cache the Parameter Buffer whereas in TCOR the cache is split to access different parts of the Parameter Buffer. The split parts have a different set-mapping and replacement policy, respectively. Secondly, owing to the cache reorganization, each line of the Attribute Cache corresponds to a primitive whereas for the baseline Tile Cache and the Primitive List Cache, it is 64 bytes of data. This makes the cost of a miss in the two L1 caches different and thus miss ratios incomparable.

To compare the effectiveness of the L1 caches, we compare the Parameter Buffer accesses made to the L2 in both cases. Figures 14 and 15 show the decrease in L2 accesses for TCOR. On average, we see around 33.5% and 37.1% decrease in the 64KiB and 128KiB experiments respectively. Note that in case of benchmarks with smaller geometry, implying a smaller Parameter Buffer, the decrease in L2 accesses goes up to 64.4%. The benchmarks with the highest average reuse of primitives (see Table II) are the ones that show a large decrease in accesses. This was expected as the higher the reuse, the more a replacement policy affects a workload.

L2. Parameter Buffer accesses to Main Memory are indirectly affected by the effectiveness of the L1 caches in this design. First of all, the decrease in the L2 cache accesses implies a possible decrease in L2 misses. Secondly, the L1 Attribute Cache has been optimized to retain attributes with the shortest reuse distance. This implies that, on average, when primitives are evicted and written to the L2, they will have lower reuse henceforth. In addition to this, the L2 replacement policy modification introduced by TCOR helps reduce the reads and writes of the Parameter Buffer into Main Memory. Figures 16 and 17 show this experimentally.

We see around 93% and 94.1% decrease in Main Memory accesses to the Parameter Buffer on average in the two experiments. In all benchmarks except for three, TCOR has been able to completely eliminate Parameter Buffer accesses to Main Memory. *DDS*'s Parameter Buffer has a memory footprint of 1.8MiB (see Table II). With a 128KiB L1 and a 1MiB shared L2 cache, it is impossible to hold the whole Parameter Buffer without making a spill to main memory. And yet we see an impressive 53-58% decrease in accesses for this benchmark. Overall, we see a huge decrease in writes because of the reduced write-backs and a dramatic decrease in reads because of the more effective replacement policy in both the L1 and L2.

Total Main Memory Accesses. As shown in Figure 2, Main Memory is accessed by the L2 cache and the Color Buffer. The L2 is accessed by the Vertex Cache, Tile Cache, Texture Caches and Instruction Caches. To evaluate the ef-

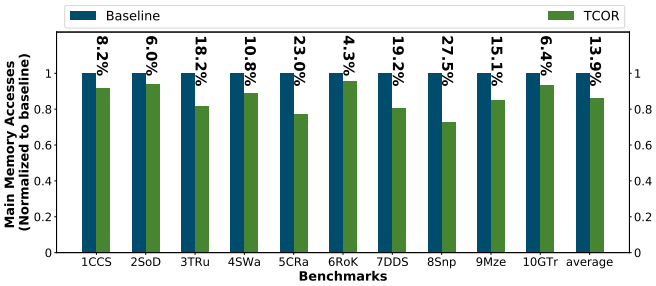

Figure 18: Total Main Memory accesses (64KiB Tile Cache).

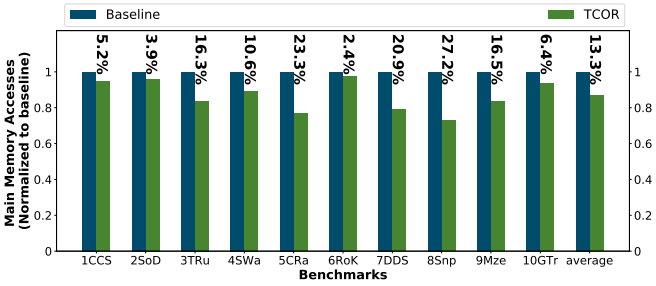

Figure 19: Total Main Memory accesses (128KiB Tile Cache).

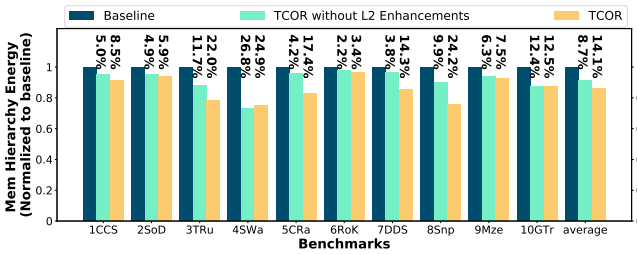

Figure 20: Energy consumed by the Memory Hierarchy (64 KiB Tile Cache).

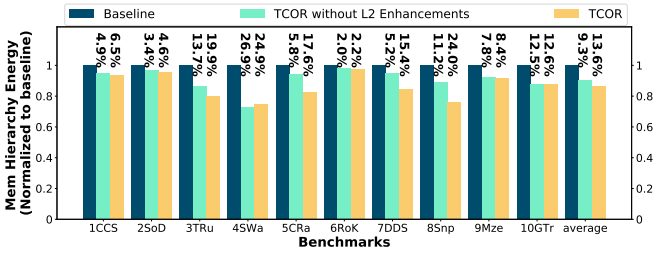

Figure 21: Energy consumed by the Memory Hierarchy (128 KiB Tile Cache).

fect of TCOR on the overall Main Memory accesses, Figures 18 and 19 plot the total amount of accesses to main memory for a 64KiB and 128KiB Tile Cache size respectively. We can see that, on average, the benchmarks exhibit a decrease of 13.9% and 13.3% in total Main Memory bandwidth. We can observe that the benchmarks with a higher geometry footprint benefit more from TCOR. In particular, CRa, DDS and Snp get the highest benefits from TCOR since these are benchmarks with more geometry and thus a larger Parameter Buffer.

2) Energy: Energy-efficiency is beneficial for all computing systems but crucial for mobile devices. Figures 20 and 21 show the energy consumption of the memory hierarchy for the baseline, TCOR without L2 enhancements and TCOR. We can see that TCOR provides around 14.1% and 13.6% decrease in the respective experiments and provides around 9% decrease without L2 enhancements. For high geometry benchmarks like Snp, we see close to 24.2% decrease in energy. This is because their Parameter Buffers are larger and constitute a larger portion of the total memory accesses in the GPU. Figure 22 plots the decrease in total GPU energy and shows a 5.6% and 5.3% decrease in the respective experiments.

3) Throughput: Our new faster Tiling Engine opens the door to a more aggressive Raster Pipeline for future works. We evaluate our new Tiling Engine independently, by modeling an experiment where we resize its output queue to have unlimited primitives, so that the Tiling Engine never has to stall because of the slower Raster Pipeline. With this, we calculate the number of primitives output by the Tile Fetcher per cycle, the maximum being 1. Figures 23 and 24 show that, on average, our technique provides a 5x speedup. This is in line with the decrease in L2 and main memory Parameter Buffer accesses reported before. Note that benchmarks like SoD come quite close to the maximum with TCOR.

VI. Related Work

Previous work on the Tiling Engine of TBR architectures focused on the computational complexity of binning, resizing the Parameter Buffer and reducing memory overhead. Our work improves the design of the memory hierarchy in the graphics pipeline to efficiently cache the Parameter Buffer.

Antochi et al. [2] explore a way to accurately calculate which tiles are overlapped by each primitive, while reducing the amount of computation. Another work [3] describes several algorithms for sorting the primitives into bins and evaluates their computational complexity and memory requirements. There is also other work [39] that aims to

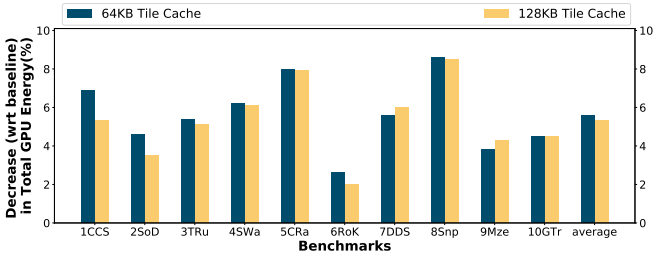

Figure 22: Decrease in Total GPU Energy wrt the baseline.

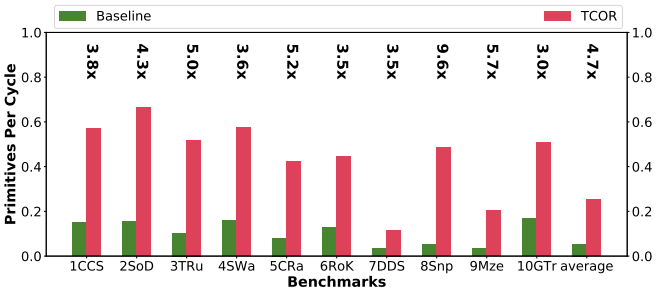

Figure 23: Primitives output per cycle by the Tile Fetcher (64KiB Tile Cache).

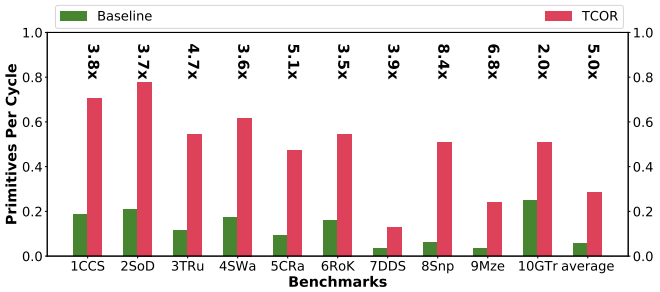

Figure 24: Primitives output per cycle by the Tile Fetcher (128KiB Tile Cache).

provide a concise tile list structure and an efficient overlap detection algorithm for mobile graphics processors to reduce the memory overhead, CPU latency and computing resources. Since large primitives may cover a significant number of tiles, they need to be recorded in the primitive lists of all related tiles. Hsiao et al. [20] propose a hierarchical primitive lists structure to minimize the size of the Parameter Buffer, list building time and data accesses to the Parameter Buffer. All the above cited works do not target the design of the memory hierarchy to improve the caching of the Parameter Buffer. Our work is novel for its approach to the memory hierarchy for the Tiling phase and most importantly for implementing an OPT replacement policy in its L1 cache.

State-of-the-art replacement policies like Hawkeye [21], Glider [16], Multiperspective [14], Perceptron [15] and DRRIP have been proven to do better than LRU for CPU workloads in the LLC (but not for L1) where the reuse pattern is filtered of temporal locality by lower-level caches. However, TCOR implements OPT for the L1 Tile Cache which caches the Parameter Buffer that has a unique access stream specific to TBR architectures.

In general, practical caches have never had optimal replacement policies although there is previous work for register allocation [19], [17]. Regardless, many implementable caches have been inspired from them. One such case is P-OPT [6] which proposes a practical OPT-inspired replacement policy for graph analytics workloads by exploiting the next-reference information encoded in the graph's representation. This work first proposes an OPT-inspired replacement policy for the LLC by accurately predicting the memory access stream of the application on a vertex granularity. However, this policy is based on the assumption that all memory accesses reach the LLC. It then uses an epoch-based lossy compression approach to store this next-reference information in some reserved ways of the LLC to realize a practical implementation of the said policy at the cost of some accuracy loss. Another inspired work is the Shepherd Cache [31] where using part of the cache for lookahead, and with that emulating OPT in the remaining cache, improves L2 performance and bridges the LRU-OPT gap to around 30-52%. Jain et al. [21] propose Hawkeye, a replacement policy that tries to look backwards over a sufficiently long history of past memory accesses to learn and mimic the optimal behavior. Over the years many more works have tried to attack the replacement problem in different ways including dead block eviction mechanisms like us [1], [8], [10], [11], [13], [18], [22]–[24], [26], [30], [32], [34], [35], [37], [38], [40]. To the best of our knowledge, this is the closest practical implementation of OPT for caches presented in literature, till date.

VII. Conclusions

In this work we introduced TCOR, a novel cache memory architecture for the Tile Cache of mobile GPUs, whose main innovation is a cost-effective implementation of the OPT replacement algorithm [27], which was formally proven to be an optimal solution for minimizing cache misses. We also implemented a new replacement policy in the L2 to amplify the effects of OPT on higher levels of the memory hierarchy.

OPT is generally deemed infeasible as it requires knowledge of future accesses. We made the observation that the memory access pattern to the Parameter Buffer of a TBR GPU architecture is built and used up by consecutive stages of the Graphics Pipeline, namely, the Polygon List Builder and the Tile Fetcher. TCOR uses information from the Polygon List Builder to infer the future accesses and then uses it for replacement in the Tile Cache. We observed that OPT minimizes misses to the same extent as an LRU cache with 6.8 times the size. We modified the L2 to identify dead cache blocks and prioritize the retention of potentially live cache blocks. This helped improve the caching capabilities of the L2 while reducing write-backs into main memory.

Experimental results showed that TCOR provides a 13.8% decrease in the energy consumed by the whole memory hierarchy. We also observed higher energy savings in benchmarks with a larger geometry footprint which allows TCOR to tackle workloads with more complex geometry in energy-constrained mobile devices. This more efficient cache architecture also translates into a higher throughput in the Tiling Engine which opens the door to more aggressive Raster Pipeline implementations, including the use of Parallel Renderers.

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] An-Chow Lai and B. Falsafi, "Selective, accurate, and timely self-invalidation using last-touch prediction," *International Symposium on Computer Architecture*, pp. 139–148, 2000.

[2] I. Antochi, B. Juurlink, S. Vassiliadis, and P. Liuha, "Efficient tile-aware bounding-box overlap test for tile-based rendering," *International Symposium on System-on-Chip*, pp. 165–168, 2004.

[3] I. Antochi, B. Juurlink, S. Vassiliadis, and P. Liuha, "Scene management models and overlap tests for tile-based rendering," *Euromicro Symposium on Digital System Design*, pp. 424–431, 2004.

[4] 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.

[5] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Teapot: A toolset for evaluating performance, power and image quality on mobile graphics systems," *ACM International Conference on Supercomputing*, p. 37–46, 2013.

[6] V. Balaji, N. Crago, A. Jaleel, and B. Lucia, "P-OPT: Practical Optimal Cache Replacement for Graph Analytics," *IEEE International Symposium on High-Performance Computer Architecture*, pp. 668–681, 2021.

[7] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[8] 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.

[9] 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.

[10] 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*, Jun. 2010.

[11] 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.

[12] A. González, M. Valero, N. Topham, and J. M. Parcerisa, "Eliminating cache conflict misses through xor-based placement functions," *International Conference on Supercomputing*, p. 76–83, 1997.

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

[14] D. Jiménez and E. Teran, "Multiperspective Reuse Prediction," *IEEE/ACM International Symposium on Microarchitecture*, pp. 436–448, 2017.

[15] E. Teran, Z. Wang and D. Jiménez, "Perceptron learning for reuse prediction," *IEEE/ACM International Symposium on Microarchitecture*, pp. 1–12, 2016.

[16] Z. Shi, X. Huang, A. Jain and C. Lin, "Applying Deep Learning to the Cache Replacement Problem," *IEEE/ACM International Symposium on Microarchitecture*, p. 413–425, 2019.

[17] Farach-Colton, Martin and Liberatore, Vincenzo, "On local register allocation," *Journal of Algorithms*, vol. 37, pp. 37–65, 2000.

[18] Gaur, Jayesh and Srinivasan, Raghuram and Subramoney, Sreenivas and Chaudhuri, Mainak, "Efficient management of last-level caches in graphics processors for 3D scene rendering workloads," *Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[19] Guo, Jia and Garzaran, Maria Jesus and Padua, David, "The power of Belady's algorithm in register allocation for long basic blocks," in *International Workshop on Languages and Compilers for Parallel Computing*, 2003, pp. 374–389.

[20] C. Hsiao, C. Chung, and H. Yang, "A hierarchical primitive lists structure for tile-based rendering," *International Conference on Computational Science and Engineering*, vol. 2, pp. 408–413, 2009.

[21] 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.

[22] 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.

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

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

[25] 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.

[26] 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.

[27] 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.

[28] P. Michaud, "Some mathematical facts about optimal cache replacement," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 4, pp. 1–19, 2016.

[29] 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.

[30] 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.

[31] K. Rajan and G. Ramaswamy, "Emulating optimal replacement with a shepherd cache," *IEEE/ACM International Symposium on Microarchitecture*, pp. 445–454, 2007.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] N. Topham and A. Gonzalez, "Randomized cache placement for eliminating conflicts," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 185–192, 1999.

[37] 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.

[38] 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.

[39] B. Yang, M. Fan, M. Han, and Y. Geng, "Optimization of false-overlap detection of tile assembly in tile-based rendering," *Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pp. 126–134, 2020.

[40] 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.