Visibility Rendering Order: Improving Energy Efficiency on Mobile GPUs through Frame Coherence

Enrique de Lucas, Pedro Marcuello, Joan-Manuel Parcerisa, Antonio González *Fellow, IEEE*

Abstract—During real-time graphics rendering, objects are processed by the GPU in the order they are submitted by the CPU, and occluded surfaces are often processed even though they will end up not being part of the final image, thus wasting precious time and energy. To help discard occluded surfaces, most current GPUs include an Early-Depth test before the fragment processing stage. However, to be effective it requires that opaque objects are processed in a front-to-back order. Depth sorting and other occlusion culling techniques at the object level incur overheads that are only offset for applications having substantial depth and/or fragment shading complexity, which is often not the case in mobile workloads. We propose a novel architectural technique for GPUs, Visibility Rendering Order (VRO), which reorders objects front-to-back entirely in hardware by exploiting the fact that the objects in graphics animated applications tend to keep its relative depth order across consecutive frames (temporal coherence). Since order relationships are already tested by the Depth Test, VRO incurs minimal energy overheads because it just requires adding a small hardware to capture that information and use it later to guide the rendering of the following frame. Moreover, unlike other approaches, this unit works in parallel with the graphics pipeline without any performance overhead. We illustrate the benefits of VRO using various unmodified commercial 3D applications for which VRO achieves 27% speed-up and 15.8% energy reduction on average over a state-of-the-art mobile GPU.

Index Terms—GPU, Graphics Pipeline, Energy-efficiency, Rasterization, Rendering, Fragment Processing, Pixel Shading, Occlusion Culling, Visibility, Tile Based Deferred Rendering, Tile Based Rendering, Topological Order.

1 Introduction

IDentifying visible surfaces is a requirement in the graphics pipeline for correct image rendering. The most widespread method to resolve visibility at pixel granularity is the Depth Test, which is typically placed at the end of the pipeline. Figure 1 introduces a simplified conventional graphics pipeline. The GPU receives vertices and processes them in the Geometry Pipeline, which generates triangles. These are then discretized by the Rasterizer, which generates fragments that correspond to pixel screen positions. Then, fragments are sent to the Fragment Processing stage, which performs the required texturing, lighting and other computations to determine their final color. Finally, the Depth test compares each fragment's depth against that already stored in the Depth Buffer to determine if the fragment is in front of all previous fragments at the same pixel position. If so, the Depth Buffer is conveniently updated with the new depth, and the color of the fragment is sent to the blending stage, which will accordingly update the Color Buffer (the buffer where the image is stored). Otherwise the fragment is discarded.

One big advantage of the Depth test is that it ensures correct scene rendering regardless of the order the opaque geometry is submitted by the CPU. The main drawback is that the color of a given pixel may be written more times than necessary (a problem known as overdraw), which wastes a considerable amount of main memory bandwidth and energy [1]. Moreover, when the GPU realizes that an object or part of it is not going to be visible, all activity required to compute its color has already been performed, with the consequent waste of time and energy (a problem known as overshading), especially in the Fragment Processor, which is the most power consuming component of the graphics pipeline [2]. Reducing the overshading produced by non-visible fragments can significantly increase the energy-efficiency of the GPU.

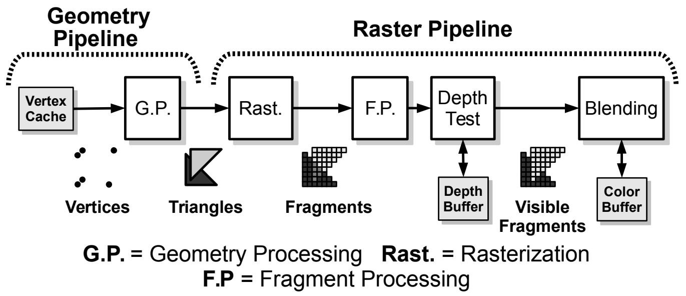

Fig. 1. Simplified version of the Graphics Pipeline.

Figure 2 shows the overshading for several applications (details on the evaluation framework are provided later). Overshading is presented here as the average number of fragments processed per pixel. First bar shows that overshading is extremely high in some applications with com-

• *Enrique de Lucas, Pedro Marcuello, Joan-Manuel Parcerisa, and Antonio González are with the Department of Computer Architecture, Polytechnic University of Catalonia (UPC), Campus Nord, Calle Jordi Girona, 1-3, 08034 Barcelona.*
E-mail: {edelucas, jmanel, antonio}@ac.upc.edu
E-mail: pedro.marcuello@gmail.com

Manuscript received July 19, 2017; revised XXXX XX, 2017.

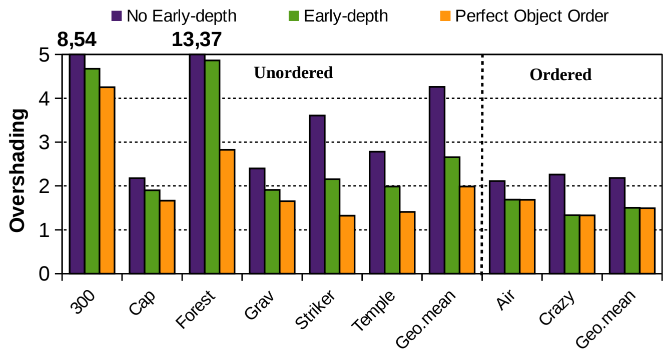

Fig. 2. Shaded fragments per pixel in a GPU without Early-depth test, with Early-depth test, and with perfect front-to-back rendering order at object granularity.

plex 3D geometry such as *300* and *Forest 2*, for which each pixel is computed and written around 8.5 and 13 times on average.

Commercial GPU pipelines include an Early-depth test stage that checks fragment visibility before the Fragment Processing, and achieves substantial overshading reductions (see mid bar of Figure 2). However, the effectiveness of the Early-depth relies on the software ability to send opaque primitives in front-to-back order, which is not what the software does in most of the cases. The third bar of Figure 2 shows the overshading with a perfect front-to-back rendering order at object level. As can be seen, there is significant headroom for improvement, and this is the target of this paper.

It is well-known that improving the battery life of handheld and portable devices is a major concern for hardware and software developers. Among all the components in smartphone SoCs, the Graphic Processing Unit (GPU) has been identified as one of the top energy consumers [3]. In particular, for graphics applications the GPU has been identified as the principal energy consumer [4]. Further experimental data with the same SoC shows a peak consumption of the GPU 50% higher than the peak consumption of the CPU [5]. The current trend towards more realistic graphics and therefore, more power hungry applications [6] is just aggravating this issue, so, improving the energy efficiency of mobile GPUs is key for future designs [7], [8], [9], [10], [11], [12], [13], [14], [15]. The development of energy-efficient solutions is a requirement to make possible a richer user experience in these platforms.

In this paper, we propose a novel hardware technique for GPUs, Visibility Rendering Order (VRO), which tries to render objects in a front-to-back order to maximize the culling effectiveness of the Early-depth test and minimize overshading, hence reducing execution time and energy consumption. Our approach is based on the observation that consecutive frames do not differ much in order to provide the feeling of smooth transition in animated applications. This suggests that the relative order among the objects in frame N is usually the same as in frame N+1 (or very close). Since depth-order relationships between objects are already checked by the Depth Test, VRO incurs minimal energy overheads because it just requires adding a small hardware to capture that information and use it later to guide the rendering of the following frame. This extra activity is performed in parallel with other stages of the pipeline, so no performance overheads are incurred.

For the analysis in this paper, we have classified our set of benchmarks into two different groups according to the following. If the reduction in overshading between an ideal front-to-back rendering order at object granularity (third bar of Figure 2) and the execution using Early-depth (second bar of Figure 2) is smaller than 0.5%, then the benchmark is categorized as ”ordered”, otherwise the benchmark is categorized as ”unordered”. Our technique achieves impressive results for the ”unordered” group of applications, i.e. those that do not submit objects in front-to-back order to the GPU. For this group, VRO obtains 27% speed-up and 15.8% energy reduction on average when compared with a state-of-the-art mobile GPU presented. For the ”ordered” group of benchmarks, VRO achieves minor reductions in overshading, but it neither produces any performance penalty nor energy overhead.

The remainder of this paper is organized as follows: Section 2 presents the visibility determination problem and how current GPUs deal with it. Our approach is described in Section 3. Section 4 discusses the implementation of our approach on a state-of-the-art GPU, and also describes the baseline GPU and a Deferred Rendering approach that will be used for comparison purposes. The experimental framework is presented in Section 5. Section 6 shows the main results of this study. The related work is discussed in Section 7 and Section 8 summarizes the main conclusions of this work.

2 Visibility Determination

As outlined in the previous section, the Depth test resolves visibility at the expense of significant overshading. Occlusion culling techniques try to reduce overshading by discarding objects completely occluded by others at the application level. Many of these software approaches require building costly spatial hierarchical data structures to render the scene from any single viewpoint. They are only effective on walkthrough applications where the entire scene is static and only the viewer moves through it, because the overheads can be amortized along a large number of frames [16]. Occlusion queries is another software technique that defines bounding volumes around dynamic objects, renders them to the GPU to test their visibility and waits for the results back to the CPU. Unfortunately, using any kind of feedback from the GPU is quite slow and limits the achievable frame rate unless the scene complexity is above a large threshold [17], which is often not the case on mobile workloads. Moreover, as mobile devices evolve to higher resolutions, testing for occlusion objects that are not fill-rate bound (i.e. with simple fragment programs and textures) may require many more pixels to fill and the GPU will likely spend more time rendering the object’s bounding volume than the object itself.

To reduce overshading, most pipelines perform an Early-depth test to fragments before they are sent to the Fragment Processors. Although the Early-depth can only cull fragments which are hidden by those already tested, it

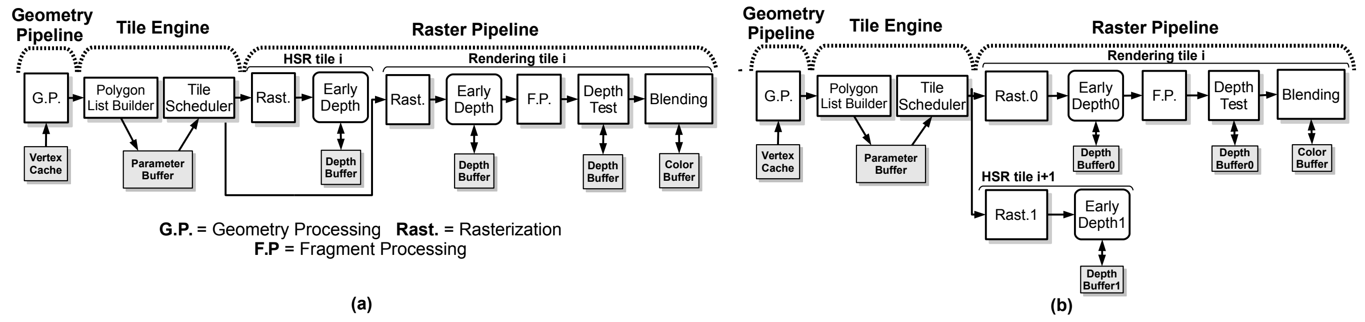

Fig. 3. Graphics pipeline: (a) Sequential DR. (b) Parallel DR.

reduces shading and blending work and brings important performance and power benefits.

Z-prepass [18] addresses overshading by performing two separate rendering passes with the GPU. First it renders the geometry without outputting to the Color Buffer, just using a null fragment shader, to setup the Depth buffer final values. On a second pass with the real shaders the Early-depth test will perform optimal culling, so overshading will be minimum (just one opaque fragment per pixel will be shaded and written to the Color Buffer). Unfortunately, this approach doubles the amount of vertex processing, rasterization and depth-test work required, which more than offset its benefits. It is only effective for workloads with enough depth and/or fragment complexity where these overheads are compensated by large fragment computation savings, which is not usually the case on mobile applications.

Like Z-prepass, Deferred Rendering (DR) is a hardware technique that avoids overshading through computing the Depth Buffer before starting fragment shading. Currently, DR has only been implemented on Tile Based Rendering (TBR) GPUs [19]. TBR pipelines divide the screen space into tiles and, before rasterization, they assign the geometry of the scene to the tiles, which are then independently rendered. This allows the GPU to use small on-chip memories to contain the Depth and the Color buffers for the entire tile, which dramatically reduces the accesses to main memory [20]. DR adds a hidden surface removal (HSR) phase to the pipeline just before the Early-Depth test. During the HSR phase, all the tile primitives are first rasterized only for position and depth, and the resultant fragments are Early-depth tested to setup the Depth Buffer. Once HSR is complete, the second pass processes the tile primitives as usual along the raster pipeline (they are read, rasterized and depth-tested again), except that this time the Early-depth test performs optimal occlusion culling. Although the exact details of this technique in commercial systems are not fully disclosed, we have modeled in our framework an efficient implementation of it at the microarchitecture level, which is described in Subsection 4.2. In contrast to Z-prepass, DR does not perform the geometry processing twice. However, as can be seen in Figure 3, DR still has a non negligible cost: either it introduces a barrier in the graphics pipeline, because the Fragment Processing stage cannot start until HSR has completely finished the tile (see (a) sequential DR), or significant extra hardware is required to perform HSR of tile *i+1* and rendering of tile *i* in parallel (see (b) parallel DR). Further details are given in Section 4.2.

3 Visibility Rendering Order

To help maximize Early-depth test effectiveness, we propose to record the visibility order of the objects in a frame, assume the same order for the next frame, and then use it to influence the rendering order of the objects in the next frame. This is expected to work since images of consecutive frames normally show a significant degree of similarity to result in a smooth transition among frames, so the ordering of objects in consecutive frames tends to be the same. To produce a quantitative evidence of this, we have evaluated sequences of 50 frames of our benchmarks, and we have observed that the relative order of the objects in a frame matches the relative order in the previous frame in more than 99% of the cases. Unlike other approaches, our technique works for all kind of scenes, either static or dynamic, it does not cause CPU-GPU synchronization issues and it has no performance cost because it works in parallel to other stages of the pipeline. This section outlines our technique, and the next section will provide hardware implementation details.

3.1 Overview

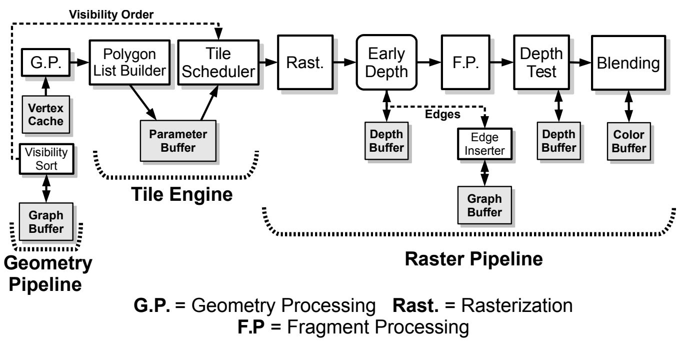

Fig. 4. Graphics pipeline including VRO.

Figure 4 shows the changes to the graphics pipeline introduced by VRO, which will be explained below: the Edge Inserter, the Visibility Sort Unit and the Graph Buffer. Basically, VRO has two stages that operate on consecutive frames:

1) **Creation of a Visibility Graph:** During the rendering of frame N the Early-depth test reveals depth precedence relationships between pairs of fragments covering the same pixel position, hence among the corresponding objects. These relationships (edges) are used by the Edge Inserter unit to

build a directed graph where objects are represented by nodes, and the edges indicate which objects are in front of others. We will refer to this graph as the Visibility Graph, which is stored in the Graph Buffer.

2) **Creation of a rendering order:** At the beginning of the rendering of frame *N+1*, in parallel with the execution of the Geometry Pipeline, the Visibility Sort unit sorts the Visibility Graph created during frame N to generate a depth-ordered list of nodes. We will refer to this list as the Visibility Rendering Order, and it is used by the Tile Scheduler to guide the rendering of the frame *N+1*.

3.2 Sort Algorithm

Once the graph is generated, it is sorted to create the Visibility Rendering Order (a front-to-back ordered list of object-ids). Our approach is based on the well known Topological Sort algorithm, first proposed by Kahn [21], which guarantees for DAG graphs (acyclic) that an ordered list of nodes exists and it is generated in linear time. Algorithm 1 outlines the basic algorithm, assuming for convenience that every node is tagged with its number of incoming edges (the in-degree). Nodes with no incoming edges are referred to as roots.

Algorithm 1 Kahn's algorithm.

```
1:  function KAHN(L,RQ)
2:          ▷ $L$: empty list that will contain the sorted nodes
3:          ▷ $RQ$: queue with all initial root nodes of $Graph$
4:      while $RQ$ is non-empty do
5:          remove head node $n$ from $RQ$
6:          insert node $n$ into list $L$
7:          for child $m$ of $n$ do
8:              remove the edge from $n$ to $m$
9:              decrease the in-degree of $m$
10:             if $m$ is a root then
11:                 add $m$ to tail of $RQ$
12:             end if
13:         end for
14:         remove node $n$ from $Graph$
15:     end while
16:     if $Graph$ has nodes then
17:         return $error$          ▷ $Graph$ has at least one cycle
18:     else
19:         return $L$              ▷ a topologically sorted order
20:     end if
21: end function
```

However, if the graph contains a cycle, this algorithm finishes with an error condition because at some point none of the graph nodes that remain to be sorted have zero in-degree. We found that these cycles are quite common, and actually none of our benchmarks creates a DAG. To cope with this situation, we distinguish three kinds of cycles and apply different solutions in each case:

1) **Auto occlusions between parts of the same object**. They are removed by discarding their corresponding edges in the process of creation of the graph, with no effect on VRO, because VRO reorders at object level and the auto-occlusions just contain intra-object precedence relations.
2) **Pairs of interlaced objects occluding each other**. VRO eliminates the cycles created by pairs of interlaced objects by adding to the Visibility Graph only the first encountered precedence relation between two objects (A, B). If later on a (B, A) relation is encountered, it is just ignored and not added to the graph.
3) **Three or more objects alternately occluding one another**. Since these cycles may be extremely costly to detect, we adopt a cost-effective approach which does not attempt to eliminate them from the graph, but it rather extends the Kahn's algorithm to side-step a cycle-induced wrong termination: whenever the *RQ* is empty and there are still nodes to be sorted but none of them is a root, our heuristic selects, from the remaining graph nodes, the next one in program rendering order with the minimum number of input edges, removes these edges from the graph, adds the node to the tail of *RQ* and iterates again. In our experiments, the heuristic is executed to select a node around 13% of the times.

Our main goal is to improve performance while still reducing energy consumption on a mobile GPU. VRO achieves both goals by improving the effectiveness of the Early-depth to cull hidden surfaces before they reach the Fragment Processing stage, so that the number of shader instructions executed is reduced. We have tested several heuristics to handle the cycles of the Visibility Graph that approximate this goal with different performance/energy trade-offs. The suitability of one or another heuristic on a given hardware platform will greatly depend on design issues that are out of the scope and space of this paper. Bear in mind however that, regardless of the approximation, image correctness is guaranteed in any case by the depth test. Our choice here is a heuristic that implies a cost effective implementation of VRO and clearly illustrates its feasibility and effectiveness.

3.3 Partial Order of Objects

The Depth Buffer only stores the depth of one fragment at every pixel position. Thus, when a new fragment is tested the comparison is performed between the new fragment and the one visible so far, so there may be objects whose fragments are never compared. Therefore, this comparisons will provide just a partial order of the objects. Hence, one may wonder whether the missing node relationships may lead to build a wrong Visibility Graph. To answer this question, note that the relative render ordering of two objects is only relevant for visibility purposes if they overlap at some region, and that region is visible at least in one pixel.

It is easy to prove that if two nodes are not connected, then either they do not overlap at all, or their overlapping region is not visible, i.e., the missing relationship is not relevant in terms of overshading. Figure 5 illustrates this property with two examples. In both cases the rendering order is A, B, C. In case (a), nodes A, C are not connected and therefore their overlapping area is not visible, so the different possible rendering orders between A and C do

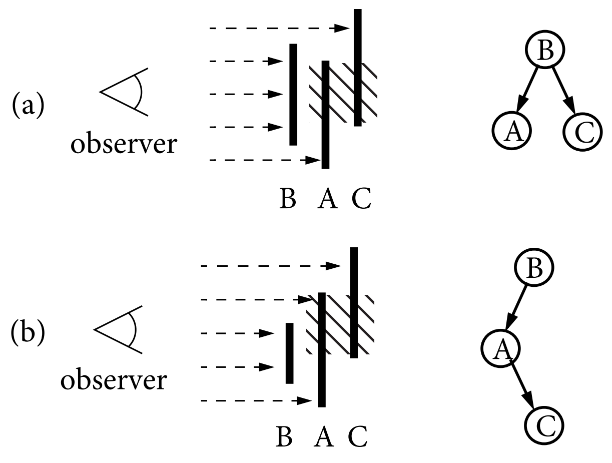

Fig. 5. Two example cases where object *B* sits in front of *A* and *C*. The shaded region highlights the overlap between *A* and *C*.

not produce a different amount of overshading. In case (b), the overlapping between A and C is partially visible and therefore these nodes are connected in the graph. That is, the partial order represented in the Visibility Graph contains all the precedence relations necessary to create an order where visible objects are scheduled before the ones that they occlude.

3.4 Visibility Rendering Order Adjustments

The Visibility Rendering Order that the Tile Scheduler receives from the Visibility Sort unit contains the object-ids of the objects rendered in the previous frame, and they may differ slightly from the objects to be rendered in the current frame. On the one hand, objects rendered in the previous frame are present in the Visibility Graph, but they are not present in the current frame and therefore they must not be scheduled, so they are simply discarded by the Tile Scheduler. On the other hand, objects not present in the Visibility Graph but present in the new frame must be scheduled, so they are put in the list after the objects in the graph.

Note that objects with Depth test disabled or with blending enabled, which are typically translucent objects, cannot be simply put at the end of the order list because it could produce erroneous images. These objects, typically rendered back-to-front after the opaque surface they overlap, must be scheduled in the same relative order as they appear in the program rendering order. VRO respects the OpenGL standard in the sense that the result is the same as if objects were processed in program rendering order, so these constrains are taken into account when creating the final rendering order. Fortunately, objects with blend enabled and with depth disabled (commonly part of the GUI of the applications) tend to be the last objects to appear in the program rendering order so they introduce minor constraints to the Visibility Rendering Order.

If the relative order among two objects changes in frame N with respect to frame N-1, VRO may introduce overshading, but in any case the correctness of the scene is guaranteed. Furthermore, in frame N+1 this ordering penalty will be corrected.

4 Implementation

This Section describes the implementation details of our technique on a contemporary GPU. We first introduce the baseline GPU. Next, we describe the extensions to the baseline architecture that are required to support Deferred Rendering (DR) and our technique (VRO). DR will be used for comparison purposes in our experiments.

4.1 Baseline TBR GPU

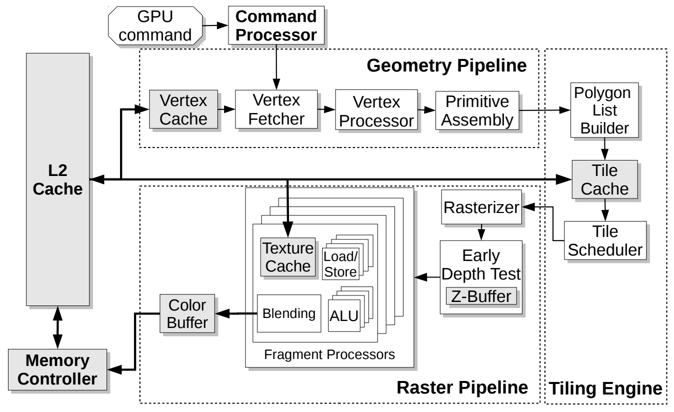

Fig. 6. Microarchitecture of a TBR GPU based on an Utgard ARM Mali.

Through the rest of the paper it is assumed a baseline TBR GPU architecture that includes one programmable Vertex Processor and four programmable Fragment Processors. Figure 6 shows a block diagram of the GPU pipeline with TBR mode. The rendering process is divided into two decoupled pipelines, Geometry and Raster, which are connected through the Tiling Engine.

The Geometry Pipeline performs geometry-related operations such as model, view and projection transformations done in the vertex processing, perspective divide, viewport transformation, clipping, and face culling. The resulting primitives are sent to the Polygon List Builder of the Tiling Engine, which assigns the primitives to tiles [20] and stores them into the Parameter Buffer, a buffer in system memory that is accessed through the Tile Cache.

Once all the primitives of a frame have been stored in the Parameter Buffer, the GPU independently processes the frame tile by tile. For every tile, the Tile Scheduler reads the primitives in program order and it sends them to the Raster Pipeline. The first component of the Raster Pipeline is the Rasterizer. The Rasterizer receives the primitives and discretizes them creating fragments that are tested in the Early-depth test (supported by an on-chip Depth Buffer). If a fragment is visible (not occluded by a previously tested fragment), it is sent to the following stages of the Raster Pipeline. Otherwise it is discarded.

The fragment colors computed in the Fragment Processors are written to the Color Buffer, which is located on-chip to reduce the main memory traffic. Once the tile rendering has been completed, the on-chip Color Buffer is flushed to the Frame Buffer in main memory. Note that for TBR pipelines, each pixel color is usually written to main memory only once regardless of object ordering. Only an overflow in the Parameter Buffer causes the GPU to render the already sorted geometry, thus writing main memory more than once, but it is highly uncommon. Note that despite the main memory bandwidth relative to the Color Buffer and the Depth Buffer are reduced with TBR mode, the geometry-related memory traffic is increased by the required communication performed to in first place store the geometry into the Parameter Buffer, and later recover it when rendering every tile.

4.2 Deferred Rendering TBR GPU

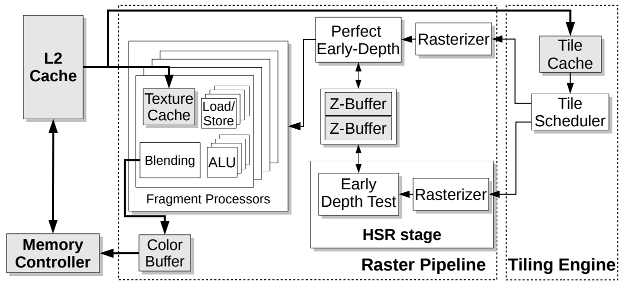

Fig. 7. Raster Pipeline of a TBR GPU implementing Deferred Rendering.

Deferred Rendering (DR) is the state-of-the-art regarding overshading reduction, so we decided to model it for comparison purposes. As outlined in Section 2, DR reduces overshading by first performing Hidden Surface Removal (HSR), which in first place computes the final state of the Depth Buffer for a given tile. Thereafter, it starts an ordinary rendering of the tile. However, given that the Depth Buffer contains the depth of the visible objects, the Early-depth is able to discard all the occluded fragments and achieve minimum overshading.

A DR technique has been commercially implemented by Imagination Technologies in their tile-based PowerVR GPU family [19], which they refer to as a TBDR. However, since only partial information about this technique has been disclosed, our Deferred Rendering implementation models what we believe is the most optimistic interpretation of this partial information, in order to be used in the comparisons with our proposal.

As previously shown in Figure 3, we developed different implementations of DR: sequential DR (a) and parallel DR (b). Sequential DR is a naïve implementation that stalls the rest of the Raster Pipeline while performing HSR. The sequential implementation badly hurts both performance and energy compared with the baseline GPU. For this scheme, the execution time increases for every one of the benchmarks tested, being 23% on average. Regarding energy consumption, it increases around 6% on average when compared with the baseline GPU. These huge overheads are due to the fact that the total time of the HSR stage (only depth rasterization plus depth test) greatly exceeds the savings provided by the overshading reduction.

Nevertheless, these huge overheads can be removed by performing the HSR stage in parallel with the other stages of the Raster Pipeline (see Figure 7). Thus, in this optimized scheme (parallel DR), while the HSR is being executed for tile *i+1*, the rest of the Raster Pipeline is executed in parallel to render the tile *i*. Obviously, this parallel implementation introduces a hardware cost and some hardware blocks, such as the Rasterizer, the Early-depth test and the Depth Buffer, need to be replicated. Furthermore, the Tile Scheduler is equipped to handle memory requests of two primitives in parallel: one primitive from the tile being rendered and the other one from the tile in the HSR stage. This does not mean that the Tile Cache has now two read ports, but the Tile Scheduler will arbitrate between both request queues and only one will be sent to the Tile Cache each cycle in a Round Robin fashion. Even though this parallel implementation of DR introduces a non negligible amount of extra hardware (6% area overhead w.r.t baseline GPU), it outperforms sequential DR in both performance and energy, so it is the one we use in the results section to be compared against our technique, VRO.

4.3 Visibility Rendering Order TBR GPU

.

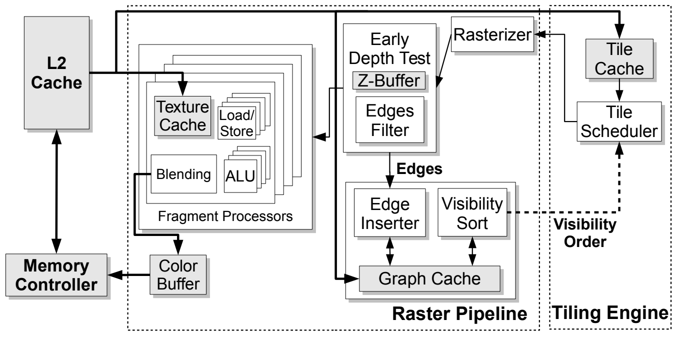

Fig. 8. Raster Pipeline of a TBR GPU implementing VRO.

As Figure 8 shows, our technique includes several new pieces of hardware: the Edge Insertion unit, the Edge Filter, the Graph Cache and the Visibility Sort unit. As usual, the control of the new hardware has been implemented using FSMs.

On the one hand, the Early-depth unit sends the edges to the Edge Inserter unit, which is responsible for storing them in the Graph Buffer, a buffer in main memory accessed through a Graph Cache and which contains the Visibility Graph of the currently rendered frame. Edge insertions take place at fragment granularity using the results of Early-Depth comparisons. However, since graph edges represent object pairs there is a large amount of tests that actually produce the same edges. The Edge Filter is a small and fast associative on-chip structure that caches the most recently inserted edges and filters out redundant insertions to the Graph Buffer, thus avoiding unnecessary Graph Cache accesses. Thanks to this structure, the Edge Insertion unit accesses the Graph Cache on average much less than once every thousand fragments.

On the other hand, the Visibility Sort unit, in parallel with the Geometry Pipeline execution of the following frame, sorts the Visibility Graph and creates a preliminary ordered list of nodes, which is sent to the Tile Scheduler. After the adequate adjustments to satisfy the restrictions presented in subsection 3.4, the Tile Scheduler produces the final Visibility Rendering Order.

Like in the baseline GPU, once the Geometry Pipeline has been executed, the Raster Pipeline renders the frame tile by tile. However, instead of reading the primitives in program rendering order the Tile Scheduler reads the primitives in Visibility Rendering Order. VRO increases the culling effectiveness of the Early-depth test and reduces overshading, which decreases the total number of instructions executed in the Fragment Processors.

In order to do a fair comparison between VRO and DR, the Tile Scheduler of VRO is also equipped with hardware to handle memory requests of two primitives of a tile in parallel. We have measured that, for the worst case (*Forest 2*), the total time required to produce a Topological Order with our version of the Kahns algorithm is almost two orders of magnitude smaller than the execution time of the Geometry Pipeline. Furthermore, the Visibility Sort unit works in parallel with other stages of the Geometry Pipeline, so negligible overheads in execution time are introduced.

4.3.1 Graph Buffer

The Graph Buffer is a small array in system memory where the Visibility Graph is stored. Due to the fact that the Visibility Graph is very sparse, it is represented as a set of adjacency lists, one per every node. Each adjacency list is implemented as a linked list of one or more entries. Each entry contains the object-id of up to W children nodes as well as other metadata shown in Figure 9.

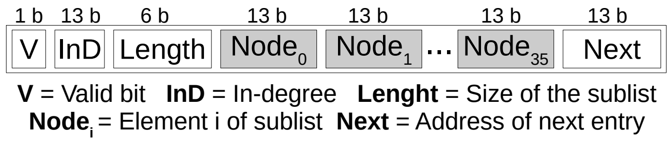
1 b
13 b
6 b
13 b
13 b
13 b
13 b
V
InD
Length
Node0
Node1
...
Node35
Next
V = Valid bit InD = In-degree Lenght = Size of the sublist
Nodei = Element i of sublist Next = Address of next entry

Fig. 9. Detail of an entry of the Graph Buffer.

VRO performs operations like membership and insertion in one step whenever the adjacency list contains less than W children nodes. Furthermore, our scheme is not constrained to a maximum of W outgoing edges per node. The adjacency lists are extended dynamically to any number of edges by allocating one or more extra overflow entries of the Graph Buffer if required. Primary lists are sequentially allocated from the lowest addresses of the buffer onwards and overflow lists are allocated from the highest addresses backwards. A buffer with N entries can store up to N nodes if none of their lists overflow, but most importantly, it can contain lists with theoretically unlimited number of edges per node.

Of course, there is a limitation imposed by the size of the memory region devoted to the Graph Buffer. However, we show that the size of the Graph Buffer represent a small region of main memory. For example, with 8192 entries and 64 B per entry the Graph Buffer is 512 KB. Figure 10 plots the amount of main memory to be allocated to the Graph Buffer for different number of maximum objects (nodes) and different number of children nodes per entry of the buffer (W). Note that even for a number of objects three orders of magnitude higher than the average number of objects observed in our set of benchmarks the memory region devoted to contain the Graph Buffer would be smaller than 11 MB. Although all our benchmarks have less than 256 nodes (see Top part of Figure 11) we provision for a much larger number of objects, so the graph has been sized to 8192 entries which is much more than enough to support common mobile workloads. Note that an object corresponds with a 3D model composed of different primitives and not a single one.

There exists a clear trade-off in the implementation of the structure of the Graph Buffer. The smaller the number of children nodes in one entry of the buffer, the smaller the total size of the Graph Buffer, but the higher the number of cycles to read the whole adjacency list of a node if it contains more than W children nodes. Figure 11 (Bottom) plots the 75th, 85th and 95th percentiles of the largest adjacency lists sizes, and it shows that most of them contain a small number of nodes (objects). For example, in the case of *300*, the values for P75, P85, and P95 mean that the 75%, 85%, and 95% of the lists contain less than 12, 17, and 27 children nodes respectively.

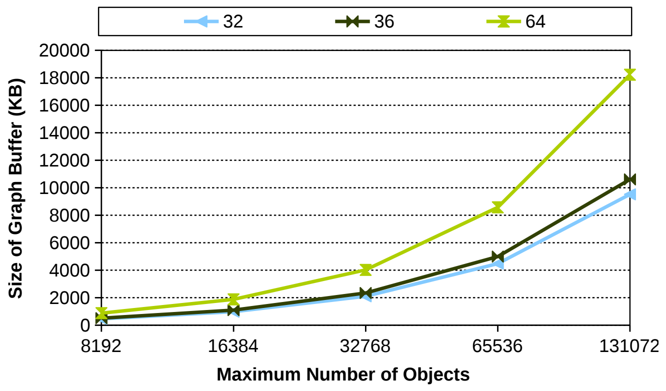

Fig. 10. Size of the Graph Buffer for different number of children nodes (W) and different number of maximum objects (from 8192 to 131072).

In the worst case, around 95% of adjacency lists of the Visibility Graph of our benchmarks have 35 or less edges per node. Hence, we allocate 36 edges per entry (W = 36), and in this way, the size of one entry is slightly smaller than 64 B and fits into a single cache block. Accounting for 8192 entries and 64 B per entry, the total size of the Graph Buffer in main memory is 512 KB. However, nothing impedes to reserve more main memory to provision for a larger number of objects. In any case, in order to reduce main memory traffic and latency, the access to the Graph Buffer is done through the Graph Cache, which is 4 KB 4-way associative.

4.3.2 Edge Inserter

The Edge Inserter is the unit responsible for creating the Visibility Graph. It works in parallel with the other stages of the Raster Pipeline. It not only creates the Visibility Graph, but also computes the in-degree of each node (see the field InD in Figure 9), which will be used by the Visibility Sort unit. The Edge Inserter (see Figure 12) receives through a queue the edges from the Early-depth test which were not filtered by the Edge Filter. The insertion of an edge (N_{src}, N_{dst}) in the Visibility Graph is a three-step process (see Figure 12):

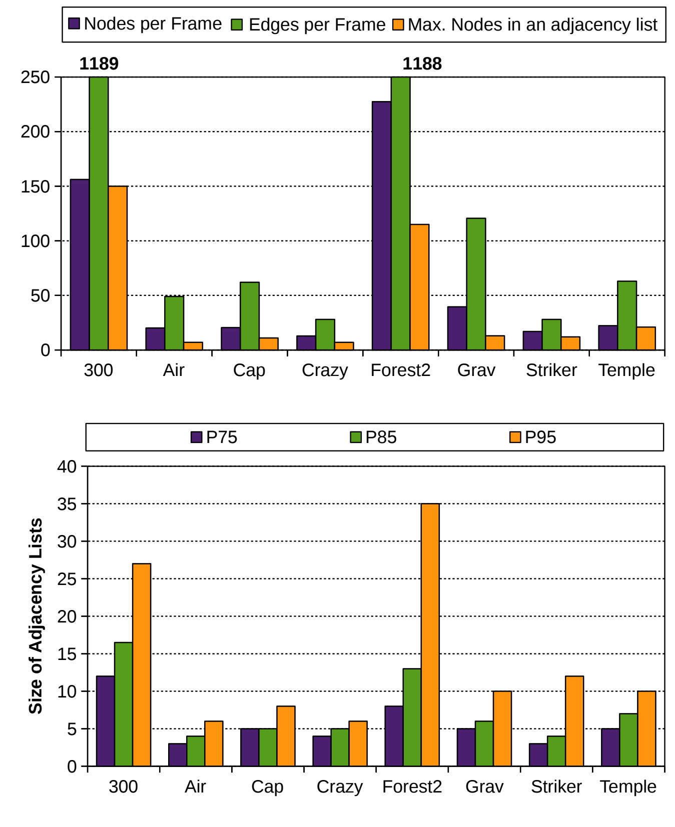

Fig. 11. (Top) Nodes per frame, edges per frame, and maximum number of nodes in an adjacency list. (Bottom) 75th, 85th and 95th percentiles of the size of the adjacency-lists of the scene graphs analyzed.

1) The primary entry of node N_{src} is read from the Graph Buffer to the AdjacencyList-Reg.
2) Then the AdjacencyList-Reg is searched to check if the edge was previously inserted in the Visibility Graph. If the primary list of node N_{src} has no linked entries (this can be done in a single clock cycle with an array of equality comparators), otherwise one or more overflow entries may be subsequently read and copied to the AdjacencyList-Reg until the edge is found or the last entry is read.
    a) If the edge already exists, then it is discarded.
    b) Otherwise, the *Length* field is increased and the edge is added to the adjacency list. However, if the entry is full, a free overflow entry is first assigned to node N_{src} and its address is stored into the *Next* field and written back to the Graph Buffer.
3) Finally, if a new edge has been added to the graph, the in-degree of the target node N_{dst} is increased.

4.3.3 Visibility Sort

The Visibility Sort unit is responsible for sorting the Visibility Graph. As outlined above, it implements an extended version of Kahn's algorithm able to handle cycles. The unit works in two phases (see Figure 13):

1) **Initial search.** The primary entries of all the nodes in the Visibility Graph are read from the Graph Buffer. If the node is a root (InD = 0), its object-id is pushed into the Roots Queue.
2) **Iterative procedure.** The object-ids stored in the Roots Queue are iteratively processed. For each node in the queue:
    a) The adjacency list of the node is read from the Graph Buffer into the AdjacencyList-Reg. At the same time, the object-id is sent to the Tile Scheduler through the Order Queue.
    b) For each child in the AdjacencyList-Reg, the in-degree is read from the Graph Buffer, and if it is still a valid node, it is decremented and written back. If the in-degree of a child becomes zero (becomes a root), it is pushed into the Roots Queue. In case of an adjacency list with overflow entries they are read in turn and processed in the same way.
    c) The node is invalidated.

If the Roots Queue becomes empty and all the graph nodes have been sorted, the algorithm finishes. Otherwise, a cycle has been found, so the unit selects the next node in program order with minimum in-degree, pushes it into the Roots Queue, and then resumes the iterative procedure. Note that the node

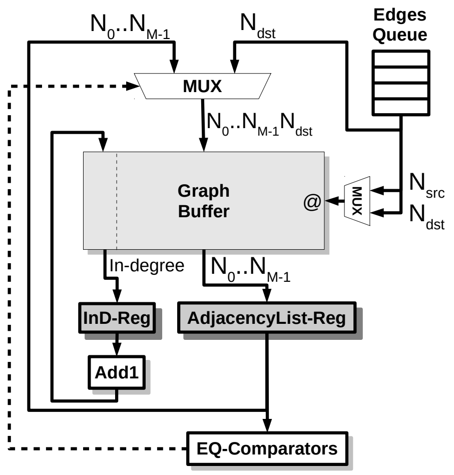

Fig. 12. Edge Insertion Hardware.

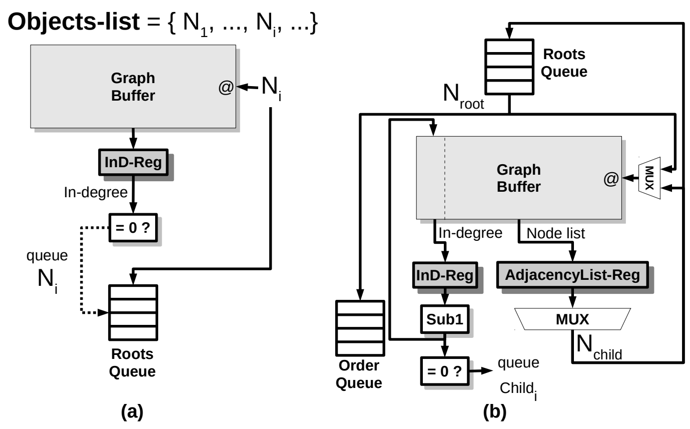

Fig. 13. Visibility Sort Hardware. (a) Initial search (b) Iterative procedure

has incoming edges remaining in the adjacency lists of its ancestors. However, since the node is invalidated after being processed, its in-degree will never be decremented again.

4.3.4 Identification of Objects

Object identifiers are required by the Visibility hardware unit to identify the objects across different frames. We need, therefore, to maintain the object identifier for objects along the graphics pipeline up to the Early-depth and the VRO unit.

A simple way to do this is to include an object identifier in every draw command of an object. This could be done using the debug marker extension of OpenGL [22], implemented in OpenGL ES 1.1 and 2.0. This extension allows the programmer of an application to annotate the OpenGL command stream with a descriptive text marker. This extension relies on the driver and the hardware to maintain the object notion through the rest of the graphics pipeline. Note that current 3D applications already uniquely identify the objects of the scene [23], so the requirement here is to pass this information from the application layer to the GPU.

5 Experimental Framework

In our experiments, we use the Teapot simulation framework [24]. We model not only the baseline GPU architecture, which closely resembles that of the Utgard microarchitecture of ARM Mali [25], but also we model Deferred Rendering and Visibility Rendering Order techniques on a TBR GPU architecture. ARM Mali Utgard microarchitecture is the most successful mobile GPU till the date, with around 19.1% of the mobile GPU market share by March 2017 [26], while TBR GPUs represent around 95% of the mobile GPU market. Despite in this work we employ a TBR GPU architecture, note that VRO is orthogonal to the TBR mode, and its implementation on top of an IMR GPU would also increase the performance and reduce the energy consumption of the GPU. Regarding our benchmarks set, it is composed of eight popular Android commercial 3D applications listed in Table 1.

TABLE 1
Benchmarks Set.

Benchmark	Alias	Description	Downloads (M)
300	300	hack & slash	10-50
Air Attack	Air	flight arcade	10-50
Captain America	Cap	beat'em up	1-5
Crazy Snowboard	Crazy	snowboard arcade	5-10
Forest 2	Forest	horror	1-5
Gravity	Grav	action	1-5
Striker	Striker	first person shooter	10-50
Temple Run	Temple	adventure arcade	100-500

TABLE 2
GPU Simulation Parameters.

Baseline GPU Parameters	
Tech Specs	400 MHz, 1 V, 32 nm
Screen Resolution	1200x768
Tile Size	16x16
Main Memory	
Latency	50-100 cycles
Bandwidth	4 bytes/cycle (dual channel)
Size	1 GB
Queues	
Vertex (2x)	16 entries, 136 bytes/entry
Triangle, Tile	16 entries, 388 bytes/entry
Fragment	64 entries, 233 bytes/entry
Caches	
Vertex Cache	64 bytes/line, 2-way associative, 4 KB, 1 bank, 1 cycle
Texture Caches (4x)	64 bytes/line, 2-way associative, 8 KB, 1 bank, 1 cycle
Tile Cache	64 bytes/line, 8-way associative, 128 KB, 8 banks, 1 cycle
L2 Cache	64 bytes/line, 8-way associative, 256 KB, 8 banks, 2 cycles
Color Buffer	64 bytes/line, 1-way associative, 1 KB, 1 bank, 1 cycle
Depth Buffer	64 bytes/line, 1-way associative, 1 KB, 1 bank, 1 cycle
Non-programmable stages	
Primitive assembly	1 triangle/cycle
Rasterizer	4 attributes/cycle
Early Z test	32 in-flight quad-fragments, 1 Depth Buffer
Programmable stages	
Vertex Processor	1 vertex processor
Fragment Processor	4 fragment processors
Extra Hardware VRO GPU	
Edges Filter	32 elements, LRU, 1 cycle
Graph Cache	64 bytes/line, 4-way associative, 4 KB, 1 bank, 1 cycle
Edge Insertion	1 Edge Inserter unit
Graph Sort	1 Visibility Sort unit
Edges Queue	64 entries, 4 bytes/entry
Order Queue	64 entries, 2 bytes/entry
Extra Hardware DR HSR stage	
Tile Queue	16 entries, 388 bytes/entry
Fragment Queue	64 entries, 233 bytes/entry
Rasterizer	4 attributes/cycle
Early Z test	32 in-flight quad-fragments, 1 Depth Buffer
Depth Buffer	64 bytes/line, 1-way associative, 1 KB, 1 bank, 1 cycle

5.1 GPU Simulation

Teapot [24] is a mobile GPU simulation infrastructure that can run unmodified commercial Android applications. It includes an OpenGL commands interceptor, a GPU trace generator and a cycle-accurate timing simulator. The parameters used in the simulations are shown in Table 2.

While a graphical application is executed in the Android emulator [27], a trace of OpenGL commands is stored. This trace is later fed to the GPU trace generator, which creates the GPU trace through the software renderer (Softpipe) included in Gallium3D [28]. The generated GPU trace file includes the Vertex Processor and Fragment Processor instructions, the memory addresses of the texture and vertex data, the primitives generated, and the corresponding fragments as well as other pipeline data required to simulate the execution. The GPU trace is fed to the cycle-accurate timing simulator, which accurately models the baseline GPU. This simulator has been extended to implement both DR and VRO GPUs as described in Section 4.

The results reported include static and dynamic energy consumption of the whole GPU, including RTL models of Edge-Insertion and Visibility-Sort, as well as the full memory hierarchy including the main memory. Teapot models the power of the GPU with McPAT [29]. Likewise, the power of the VRO unit has been modeled using McPAT's components, shown between parenthesis in the following list: Graph Cache (Cache); EQ Comparators (XOR); Muxes (MUX); Min-Comparator (ALU); Adders (ALU); Subtractors (ALU); and registers. The area overhead of VRO is less than 1% whereas for DR it is around 6% (w.r.t. baseline TBR in both cases).

6 Experimental Results

In this section we present the performance and energy savings of VRO with respect to the baseline TBR GPU. Furthermore, the benefits of VRO are compared with those of DR.

6.1 Effectiveness of VRO

Figure 14 shows the normalized speed-up achieved by our technique (VRO) and by DR relative to the ARM Mali-like baseline TBR GPU. As it can be observed, VRO achieves up to 1.42x speed-up (*Forest 2*), and 1.27x on average, being the lowest speed-up 1.14x (*Captain America*). DR achieves up to 1.25x speed-up (*Striker*), and 1.17x on average, being the lowest speed-up 1.13x (*Gravity*). Regarding system energy (see Figure 15), the consumption of VRO is reduced up to 0.76x (*Forest 2*) and 0.84x on average, being the lowest reduction 0.91x (*Captain America*). DR reduces it up to 0.82x (*Forest 2*) and 0.88x on average, being the lowest reduction 0.96x (*300*). Recall that for sequential DR (not included in the graph), the execution time increases for every one of the benchmarks tested, 23% on average, while the energy consumption increases around 6% on average when compared with the baseline GPU.

The performance advantage of VRO with respect to DR is the result of several factors. On the one hand, DR may reduce more overshading than VRO because it works at pixel granularity whereas VRO reorders the geometry at object granularity. The extra fragments processed by VRO may induce pipeline stalls and hurt performance only in case that they fill the queue that feeds the Fragment Processors. On the other hand, the Tile Scheduler of DR reads in parallel primitives of tile i+1 for the HSR unit, and primitives of tile i for the conventional Raster Pipeline, which may increase latency and starve the Fragment Processors. For an equally sized available Tile Cache bandwidth, DR produces substantially more accesses to the cache (which may degrade throughput) and has a larger working set (which may degrade miss rate and latency). To show the relative importance of these factors, we have measured both the overshading and the average fetch time to the Tile Cache.

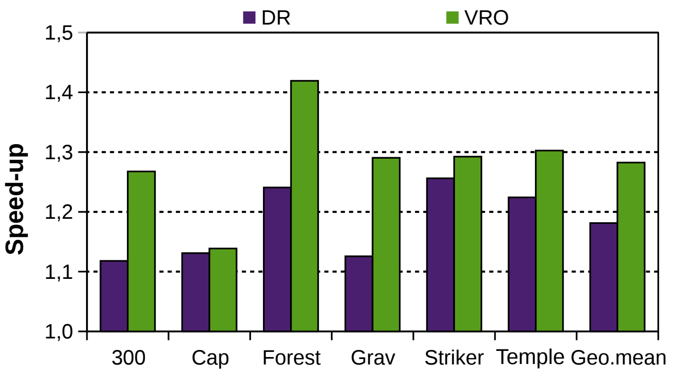

Fig. 14. Speed-up of DR and VRO normalized to the baseline TBR GPU.

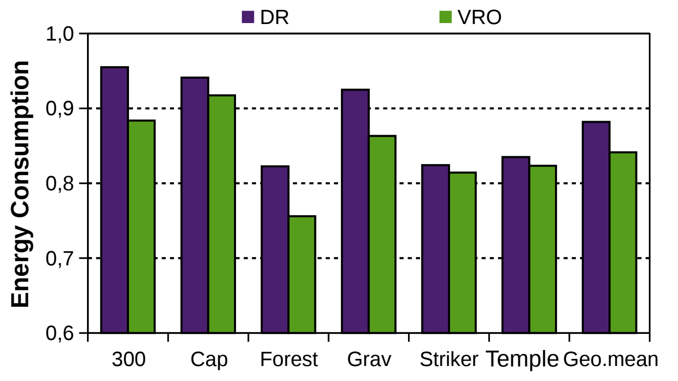

Fig. 15. Energy consumption of DR and VRO normalized to the baseline TBR GPU.

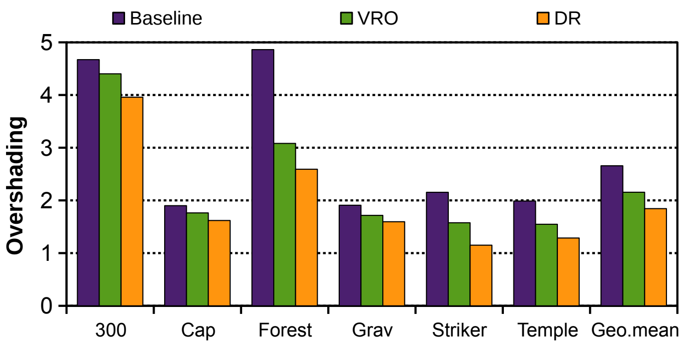

Fig. 16. Overshading of the baseline TBR GPU, DR and VRO.

Figure 16 plots the overshading for DR and VRO compared to the overshading of the baseline GPU. As expected, the overshading with DR (close to 0.7x w.r.t. baseline) is smaller than the overshading with VRO (close to 0.81x w.r.t.

baseline). The smaller overshading reduction of VRO is mainly caused by the fact that DR performs the HSR stage at pixel level granularity, while VRO performs the sorting at object-level granularity. Nonetheless, the figure shows that VRO consistently achieves significant overshading reductions for all the benchmarks, in line with the speedups reported in Figure 14.

Figure 18 plots the average fetch time per primitive in cycles for DR and VRO. It shows substantial fetch time increases for DR with respect to VRO. On average, the fetch time for DR is 68 cycles while it is only 50 cycles for VRO (25% less). Furthermore, the number of primary misses of the Tile Cache is 11% higher for DR than for VRO.

Figure 17, shows the normalized memory-traffic of VRO and DR w.r.t. baseline GPU. As can be seen the bandwidth of VRO and DR is 0.98 and 0.96 respectively. On the one hand, because of its lower overshading, DR saves more texture traffic than VRO. But on the other hand, DR must read twice the number of primitives to execute the HSR phase, which increases main memory traffic. Regarding the extra accesses of VRO to the Visibility-Graph, they add less than a tiny 0.005% to the total memory traffic.

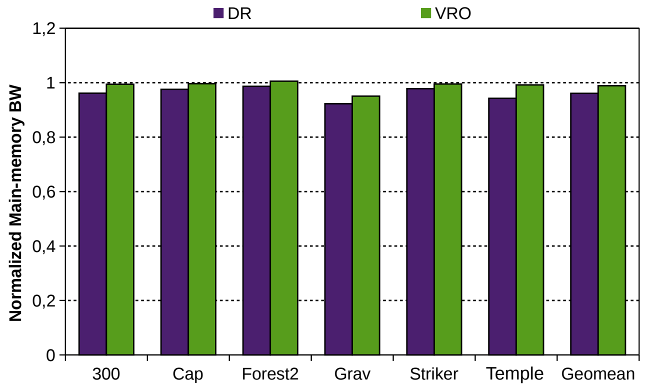

Fig. 17. Normalized memory traffic of DR and VRO w.r.t. baseline GPU.

Figure 19 compares the relative importance of the above two factors and explains why VRO outperforms DR. It plots the absolute time difference in cycles of DR with respect to VRO, for different parameters: total number of cycles to fetch primitives (first bar), total number of pipeline stall cycles caused by Fragment Processor input queue full (second bar), and total execution time (third bar). The first bar shows that DR spends many more cycles than VRO to fetch primitives, more than 27 Million cycles on average, which is caused by the higher latencies reported in Figure 18. The second bar shows that DR experiences less stall cycles caused by busy Fragment Processors, about 2.85 Million cycles less than VRO. This is related to the better overshading reduction of DR reported in Figure 16. Note however that not all the extra fragments of VRO cause a pipeline stall, only in case that they fill the queue that feeds the Fragment Processors.

The third bar is not just the sum of the other two factors. Not all the extra fetch cycles incurred by DR are ultimately translated to net increases of the execution time, because the buffers in between the Raster Pipeline stages partially smooth the effect of the initial fetching overheads. The increment in fetching time is ultimately translated to an increment in execution time around 21.6 Million cycles on average, as expected from the large difference between the other two bars. It explains the speed-ups reported in Figure 14.

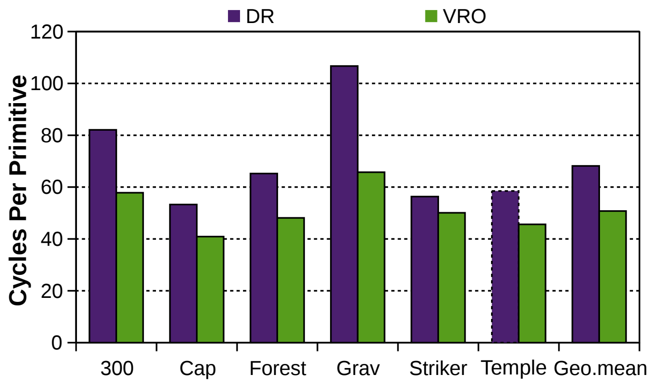

Fig. 18. Number of cycles to read a primitive with DR and VRO.

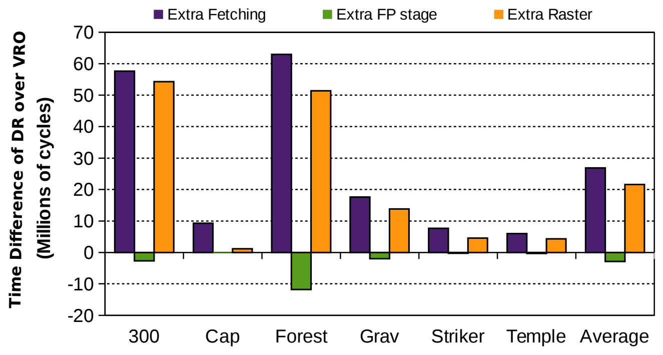

Fig. 19. Increment of cycles reading geometry (first bar), increment of stall cycles caused by the Fragment Processing stage (second bar), and increment of cycles of execution of the Raster Pipeline (third bar) all using DR w.r.t. VRO.

In benchmarks such as *300*, *Forest 2* and *Gravity* the extra fetching cycles are largely translated into extra execution time of the Raster Pipeline. This is because these benchmarks have around one order of magnitude more primitives than the other ones, which means that the overhead of reading the geometry relative to the total time of the Raster Pipeline is greater than in the other benchmarks. Furthermore, these benchmarks have less fragments per primitive than the others. The smaller the number of fragments per primitive the faster the queue that feeds the Rasterizer gets empty.

In the case of *Captain America*, the initial increment of the fetching cycles is hardly reflected as an overhead in the total processing time of the Raster Pipeline. Unlike other benchmarks (*300*, *Forest 2* and *Gravity*), *Captain America* has a much lower number of primitives and a greater number of fragments per primitive.

In conclusion, we have shown that even reducing less overshading than DR, VRO achieves higher speed-up because the overhead in fetch cycles with DR is much higher than the overhead caused by the extra fragment processing with VRO. Moreover, take into account that the area overhead of VRO is less than 1% whereas the area overhead of DR is around 6%. Therefore, some of this area could be used to implement more complex schemes of VRO in order to further reduce overshading. The overshading can be differentiated in two types: intra-object and inter-object overshading. The former is produced by auto-occlusions of an object. The latter is the overshading caused by occlusions between different objects. Given that VRO sorts at object-level granularity, it is only reducing inter-object overshading. However, as we show in the related work section below, there are techniques which are complementary to VRO and that effectively reduce the intra-object overshading. Hence, we believe that VRO has still room for improving performance and energy savings by combining it with one of those techniques.

7 Related work

The impact of overshading has been thoroughly studied in the past. Olson [1] studies such effect in mobile platforms for a set of commercial mobile applications and identifies overshading as a significant source of wasted energy.

Multiple works have analyzed how overshading can be reduced by culling the geometry at primitive level granularity through the use of occlusion queries [30], [31]. When using occlusion queries, the application usually sends a query with a Bounding Volume of the object to the GPU to be rasterized and depth-tested. Eventually, the result of the query is sent to the driver and if the Bounding Volume was occluded, the application will not send the object to the GPU. Given that at some point the application must know the result of the queries, they may introduce CPU stalls and produce GPU starvation. Furthermore, some drivers let the GPU render several frames behind the CPU by actually queuing the rendering commands [32], [33], [34], which exacerbates these problems. Furthermore, like the Early-depth, occlusion queries require to sort the queries (and the objects) front-to-back to perform well. On the contrary, VRO does not suffer by these limitations. On the one hand VRO is fully integrated into the GPU, so the application does not need to receive any feedback from VRO. On the other hand, VRO reuses the results produced in the Depth test of the actual rendering commands of the application, instead of introducing extra work (occlusion queries) to reduce overshading.

Govindaraju et al. [35] sort the primitives of every object of a scene in a front-to-back order from a given viewpoint. However, they assume that the objects do not overlap, so the scheme only avoids intra-object overshading. Furthermore, they are not able to handle cycles in the sorting process whereas VRO is able to produce a Visibility Rendering Order in the presence of cycles, which are highly common on 3D scenes. There are other approaches focused on reducing intra-object overshading. Nehab et al. [36] and Sander et al. [37], propose a pre-processing scheme that sorts the triangles in a view-independent order to reduce overdraw. However, they focus on static meshes and produce a single order per mesh. On the other hand, Han et al. [38] target animations and produce different view-dependent orders per object, which are used by the application depending on the orientation of the objects respect to the camera. These techniques, focused on reducing intra-object overshading are complementary to VRO, which reduces inter-object overshading.

Like z-prepass, conventional Deferred Rendering [39], [40] avoids to execute occluded geometry in the Fragment Processors. Clarberg et al. [41] propose a Deferred Rendering method that is executed in two phases. The first one rasterizes the geometry of the scene and stores some intermediate results. The second phase sorts the data into tiles, and then for each tile repeats the vertex processing for the visible primitives. The authors only report bandwidth and Fragment Processor executions and no energy numbers, which may be high due to the extra activity. Imagination Technologies implements a Deferred Rendering approach in its family of GPUs Power VR [19].

Arnau et al. [42] exploit frame coherence to reduce overshading. They render two frames in parallel and introduce a memoization scheme that caches results of the Fragment Processors to avoid redundant executions, rather than discard hidden fragments. They reduce redundant computations (visible or not), while VRO culls hidden fragments (redundant or not). Both techniques are complementary because they focus on different problems.

Rendering the objects in a front-to-back order effectively reduces overshading but unfortunately it is not the general case in commercial applications. Our proposal creates a view-dependent front-to-back order that effectively reduces overshading in a transparent manner to the programmer. VRO does not require neither extra Vertex Processing, Rasterization nor Early-depth executions. Furthermore, VRO can handle both static and animated scenes and is able to create a rendering order of a scene even in the presence of cycles between different objects.

8 Conclusion

In this paper we have presented VRO, a novel technique that effectively reduces overshading. VRO is based on the observation that the relative order among the objects of a scene tends to be very similar between one frame and the next. VRO includes a small hardware unit that stores the order relations among the objects of a scene of the current frame in a buffer. This information is used in the next frame, while the GPU is executing in parallel the Geometry Pipeline, to create a Visibility Rendering Order that guides the Tile Scheduler. The overhead of this technique is minimum, requiring less than 1% of the total area of the GPU while its latency is hidden by other processes of the graphics pipeline.

For a set of unmodified commercial applications, VRO outperforms state-of-the-art techniques in performance and energy consumption by reducing the overshading without the need of a expensive HSR stage at fragment granularity. VRO is especially efficient for geometry-complex applications, which are expected to be the most common applications in mobile devices as they already are in desktops. VRO achieves a speed-up about 1.27x and an energy consumption around 0.85x compared to an ARM Mali-like GPU. VRO outperforms DR because the Visibility Rendering Order is created out of the critical path while DR introduces significant overheads to perform the HSR stage.

Acknowledgments

This work has been supported by the Spanish State Research Agency under grants TIN2013-44375-R, TIN2016-75344-R (AEI/FEDER, EU), and BES-2014-068225.

References

[1] T. J. Olson, "Hardware 3d graphics acceleration for mobile devices," in *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, March 2008, pp. 5344–5347.

[2] J. Pool, "Energy-precision tradeoffs in the graphics pipeline," dissertation, The University of North Caroline at Chapel Hill, 2012.

[3] X. Chen, Y. Chen, Z. Ma, and F. C. A. Fernandes, "How is energy consumed in smartphone display applications?" in *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '13. New York, NY, USA: ACM, 2013, pp. 3:1–3:6. [Online]. Available: http://doi.acm.org/10.1145/2444776.2444781

[4] S. Patil, Y. Kim, K. Korgaonkar, I. Awwal, and T. S. Rosing, *Characterization of User's Behavior Variations for Design of Replayable Mobile Workloads*. Cham: Springer International Publishing, 2015, pp. 51–70. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-29003-4_4

[5] (accessed July 10, 2017) Qualcomm snapdragon s4 (krait) performance preview. http://www.anandtech.com/show/5559/qualcomm-snapdragon-s4-krait-performance-preview-msm8960-adreno-225-benchmarks/4.

[6] S. He, Y. Liu, and H. Zhou, "Optimizing smartphone power consumption through dynamic resolution scaling," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '15. New York, NY, USA: ACM, 2015, pp. 27–39. [Online]. Available: http://doi.acm.org/10.1145/2789168.2790117

[7] T. Akenine-Moller and J. Strom, "Graphics processing units for handhelds," *Proc. of the IEEE*, vol. 96, no. 5, pp. 779–789, May 2008.

[8] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Boosting mobile gpu performance with a decoupled access/execute fragment processor," in *Comp. Archit. (ISCA), 2012 39th Annual Int. Symp. on*, June 2012, pp. 84–93.

[9] S.-L. Chu, C.-C. Hsiao, and C.-C. Hsieh, "An energy-efficient unified register file for mobile gpus," in *Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th Int. Conf. on*, Oct 2011, pp. 166–173.

[10] J. Hasselgren and T. Akenine-Möller, "Efficient depth buffer compression," in *Proc. of the 21st ACM SIGGRAPH/EUROGRAPHICS Symp. on Graphics Hardware*, ser. GH '06. New York, NY, USA: ACM, 2006, pp. 103–110.

[11] B. Mochocki, K. Lahiri, and S. Cadambi, "Power analysis of mobile 3d graphics," in *Proc. of the Conf. on Design, Automation and Test in Europe: Proc.*, ser. DATE '06. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 502–507.

[12] V. Moya, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa, "Shader performance analysis on a modern gpu architecture," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 38. Washington, DC, USA: IEEE Computer Society, 2005, pp. 355–364. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2005.30

[13] J. Rasmusson, J. Hasselgren, and T. Akenine-Möller, "Exact and error-bounded approximate color buffer compression and decompression," in *Proc. of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symp. on Graphics Hardware*, ser. GH '07. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, pp. 41–48.

[14] E. de Lucas, P. Marcuello, J.-M. Parcerisa, and A. González, "Ultra-low power render-based collision detection for cpu/gpu systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 445–456. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830783

[15] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally, "Architecting an energy-efficient dram system for gpus," in *23rd International Symposium on Higher Performance Computer Architecture*, ser. HPCA, 2017.

[16] D. Cohen-Or, Y. L. Chrysanthou, C. T. Silva, and F. Durand, "A survey of visibility for walkthrough applications," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 3, pp. 412–431, Jul. 2003. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2003.1207447

[17] D. Bartz, M. Meiner, and T. Httner, "Opengl-assisted occlusion culling for large polygonal models," 1999.

[18] E. Haines and S. Worley, "Fast, low memory z-buffering when performing medium-quality rendering," *J. Graph. Tools*, vol. 1, no. 3, pp. 1–6, Feb. 1996. [Online]. Available: http://dx.doi.org/10.1080/10867651.1996.10487459

[19] (accessed July 10, 2017) Powervr. http://www.imgtec.com/powervr/powervr-architecture.asp.

[20] I. Antochi, *Suitability of Tile-based Rendering for Low-power 3d Graphics Accelerators*. Universitatea Politehnica Bucureşti, 2007.

[21] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962. [Online]. Available: http://doi.acm.org/10.1145/368996.369025

[22] S. Sowerby and B. Lipchak. (accessed March, 2017) Ext_debug_marker. https://www.khronos.org/registry/gles/extensions/EXT/EXT_debug_marker.txt.

[23] J. Gregory, *Game Engine Architecture, Second Edition*, 2nd ed. Natick, MA, USA: A. K. Peters, Ltd., 2014.

[24] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Teapot: A toolset for evaluating performance, power and image quality on mobile graphics systems," in *Proc. of the 27th Int. ACM Conf. on Int. Conf. on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 37–46.

[25] (accessed July 10, 2017) Mali-400 mp: A scalable gpu for mobile devices. http://www.highperformancegraphics.org/previous/www_2010/media/Hot3D/HPG2010_Hot3D_ARM.pdf.

[26] (2017) Hardware gpu market. http://hwstats.unity3d.com/mobile/gpu.html.

[27] (accessed July 10, 2017) Android studio. https://developer.android.com/studio/intro/index.html.

[28] (accessed July 10, 2017) Gallium3d. https://www.freedesktop.org/wiki/Software/gallium/.

[29] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "The mcpat framework for multicore and manycore archit.s: Simultaneously modeling power, area, and timing," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 1, pp. 5:1–5:29, Apr. 2013.

[30] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer, "Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful," *Computer Graphics Forum*, 2004.

[31] D. Sekulic, "Efficient occlusion culling," in *GPU Gems*, Nvidia, Ed., 2004, pp. 487–503.

[32] P. Cozzi and C. Riccio, *OpenGL Insights pp. 396, 417, 494*. CRC Press, July 2012, http://www.openglinsights.com/.

[33] (accessed July 10, 2017) Stuttering in game graphics: Detection and solutions. https://developer.nvidia.com/sites/default/files/akamai/gameworks/CN/Stuttering_Analysis_EN.pdf.

[34] (accessed July 10, 2017) The mali gpu: An abstract machine, part 1 - frame pipelining. https://community.arm.com/groups/arm-mali-graphics/blog/2014/02/03.

[35] N. K. Govindaraju, M. Henson, M. C. Lin, and D. Manocha, "Interactive visibility ordering and transparency computations among geometric primitives in complex environments," in *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, ser. I3D '05. New York, NY, USA: ACM, 2005, pp. 49–56. [Online]. Available: http://doi.acm.org/10.1145/1053427.1053435

[36] D. Nehab, J. Barczak, and P. V. Sander, "Triangle order optimization for graphics hardware computation culling," in *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, ser. I3D '06. New York, NY, USA: ACM, 2006, pp. 207–211. [Online]. Available: http://doi.acm.org/10.1145/1111411.1111448

[37] P. V. Sander, D. Nehab, and J. Barczak, "Fast triangle reordering for vertex locality and reduced overdraw," in *ACM SIGGRAPH 2007 Papers*, ser. SIGGRAPH '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1275808.1276489

[38] S. Han and P. V. Sander, "Triangle reordering for reduced overdraw in animated scenes," in *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '16. New York, NY, USA: ACM, 2016, pp. 23–27. [Online]. Available: http://doi.acm.org/10.1145/2856400.2856408

[39] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt, "The triangle processor and normal vector shader: A vlsi system for high performance graphics," *SIGGRAPH Comput.*

Graph., vol. 22, no. 4, pp. 21–30, Jun. 1988. [Online]. Available: http://doi.acm.org/10.1145/378456.378468

[40] T. Saito and T. Takahashi, “Comprehensible rendering of 3-d shapes,” *SIGGRAPH Comput. Graph.*, vol. 24, no. 4, pp. 197–206, Sep. 1990. [Online]. Available: http://doi.acm.org/10.1145/97880.97901

[41] P. Clarberg, R. Toth, and J. Munkberg, “A sort-based deferred shading architecture for decoupled sampling,” *ACM Trans. Graph.*, vol. 32, no. 4, pp. 141:1–141:10, Jul. 2013. [Online]. Available: http://doi.acm.org/10.1145/2461912.2462022

[42] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, “Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization,” in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA ’14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 529–540. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665748

Enrique de Lucas received B.S. degree in Computer Science in 2010 and M.S. degree in Computer Engineering in 2011 both from Universidad Complutense de Madrid (UCM), Spain. In 2011 he was Research Assistant in ArTecs research group (UCM). During 2012 he was research intern at Intel Barcelona Research Center focusing on processor micro-architecture. By February 2013 he joined ARCO research group of Universitat Politècnica de Catalunya (UPC), Barcelona (Spain), where he received the PhD degree by 2018. His main research interests include techniques to exploit inter-frame coherency and redundancy of the graphics subsystem for increase the energy-efficiency of GPUs. By January 2017, he joined a Stealth Mode Startup Company, where he holds an R&D Computer Architect position with a special focus on RISC-V processors.

Pedro Marcuello received the bachelor’s and PhD degrees in computer science from the Universitat Politècnica de Catalunya (UPC), Barcelona, in 1995 and 2003, respectively. From 2003 to 2014, he has been with the Intel-UPC Barcelona Research Center as a research scientist. From 1997 to 2003, he was with the Department of Computer Architecture, UPC, as a full-time teaching assistant. His research interests include research and development of hardware architectures, hybrid software/hardware co-designed processors, multi-threaded and multi-core processors, graphic processors, and software development for a range of designs from high performance computing to ultra-low power designs.

Joan-Manuel Parcerisa received his M.S. and PhD. degrees in Computer Science from the Universitat Politècnica de Catalunya (UPC), in Barcelona, Spain, in 1993 and 2004 respectively. Since 1994 he is a professor at the Computer Architecture Department in the Universitat Politècnica de Catalunya. His research topics include decoupled access/execute architectures, clustered microarchitectures, predication for O-o-O execution, cache memories, and ultra-low power GPU architectures for mobile devices.

Antonio González (PhD 1989) is a Full Professor at the Computer Architecture Department of the Universitat Politècnica de Catalunya, Barcelona (Spain), and the director of the Microarchitecture and Compilers research group. He was the founding director of the Intel Barcelona Research Center from 2002 to 2014. His research interests focus on computer architecture and code optimization. Antonio is an IEEE Fellow.