Omega-Test: A Predictive Early-Z Culling to Improve the Graphics Pipeline Energy-Efficiency

David Corbalán-Navarro, Juan L. Aragón, *Member, IEEE*, Martí Anglada, Enrique de Lucas, Joan-Manuel Parcerisa and Antonio González, *Fellow, IEEE*

Abstract—The most common task of GPUs is to render images in real time. When rendering a 3D scene, a key step is to determine which parts of every object are visible in the final image. There are different approaches to solve the visibility problem, the Z-Test being the most common. A main factor that significantly penalizes the energy efficiency of a GPU, especially in the mobile arena, is the so-called *overdraw*, which happens when a portion of an object is shaded and rendered but finally occluded by another object. This useless work results in a waste of energy; however, a conventional Z-Test only avoids a fraction of it. In this paper we present a novel microarchitectural technique, Omega-Test, to drastically reduce the overdraw on a Tile-Based Rendering (TBR) architecture. Graphics applications have a great degree of inter-frame coherence, which makes the output of a frame very similar to the previous one. The proposed approach leverages the frame-to-frame coherence by using the resulting information of the Z-Test for a tile (a buffer containing all the calculated pixel depths for a tile), which is discarded by nowadays GPUs, to predict the visibility of the same tile in the next frame. As a result, Omega-Test early identifies occluded parts of the scene and avoids the rendering of non-visible surfaces eliminating costly computations and off-chip memory accesses. Our experimental evaluation shows average EDP savings in the overall GPU/Memory system of 26.4% and an average speedup of 16.3% for the evaluated benchmarks.

Index Terms—Graphics processors, Mobile processors, Portable devices, Hardware architecture, Processor architecture, Energy-aware systems, Low-power design, Hidden line/surface removal, Visibility determination.

1 Introduction

Mobile devices, such as smartphones, tablets or smartwatches, have undergone a major evolution over the recent years. Users increasingly demand more complex applications on these devices, which requires higher performance at similar energy consumption. As a consequence, the energy efficiency is one of the most important aspects in mobile devices [1], [2], especially in graphics applications such as 3D games, for which the visual quality, richer graphics details, higher screen resolutions, and smooth movements are crucial for the best user experience.

Graphics workloads are generally executed in the GPU, which draws a frame of a scene by executing a set of configuration commands followed by a set of draw commands. When executing a draw command, the GPU first reads a set of vertices that model the geometry in a 3D space and applies some transformations to combine them into *primitives* (commonly triangles) projected on the screen space. Primitives are then discretized by the Rasterizer into elements known as *fragments*, for which a color in the screen is computed by using user-defined programs (a fragment shader) that, among other operations, apply a shading and a lighting model and map textures.

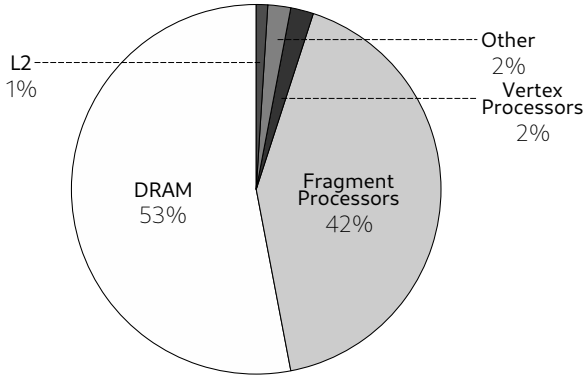


Fig. 1. Power breakdown for the baseline GPU used in our evaluation.

Previous studies [3], [4] identify the GPU as one of the most energy-consuming components on current SoCs due to the large number of computations and memory accesses needed to render a scene. Figure 1 shows the power breakdown for a conventional Tile-Based Rendering (TBR) architecture, widely adopted in mobile GPUs [5]. In particular, both the accesses to main memory and the activity of the Fragment Processors are by far the two major contributors, responsible for 53% and 42%, respectively, of the overall GPU power dissipation whereas the Vertex Processors incur a very minor energy consumption (2%) [6]. This is not surprising since the fragments processed in a scene by the Fragment Processors commonly outnumber the amount of primitives by two orders of magnitude (our experimental results show a ratio of 125:1 for the evaluated benchmarks).

- *David Corbalán-Navarro and Juan L. Aragón are with the Dept. de Ingeniería y Tecnología de Computadores, Universidad de Murcia, Spain. E-mail: {dcorbalan, jlaragon}@ditec.um.es*
- *Enrique de Lucas was with Esperanto Technologies, Mountain View, CA 94040, United States. He is now with Imagination Technologies, Imagination House, Kings Langley, WD4 8LZ, United Kingdom. E-mail: enrique.delucas@imgtec.com*
- *M. Anglada, J.-M. Parcerisa and A. González are with the Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Spain. E-mail: {manglada, jmanel, antonio}@ac.upc.edu*

Manuscript received XXX; revised YYY.

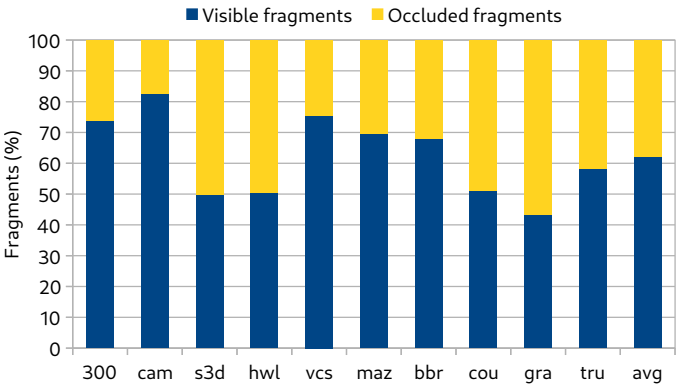


Fig. 2. Processed fragments broken down into visible and occluded ones for the evaluated benchmarks. Occluded fragments represent the overall amount of overdraw, and hence, a waste of resources.

Given the huge number of fragments to be processed in every frame and the high computational cost of rendering a single fragment, it is crucial not to waste precious resources on shading fragments that will be later occluded by other primitives. For that reason, the *visibility determination* is a fundamental task of the graphics pipeline in order to detect visible and occluded surfaces [7]. In particular, fragments that appear behind others (for a given camera viewpoint) are not visible in the final scene. The solution to the visibility problem is not unique and multiple approaches can be found in the literature [8], [9], [10] being the so-called Depth Test (or Z-Test) [11], which performs a visibility test at the pixel granularity, the most widely implemented technique in contemporary GPUs.

While the Depth Test ensures the correct visibility determination regardless of the order in which the scene is processed, it is often inefficient in the sense that each pixel in the final screen may be rendered multiple times. This problem is usually referred to as *overdraw*, which is very common in games with complex or poorly optimized scenes. Overdraw occurs when more than one opaque fragment is drawn in the same pixel position on the screen, as only the one closest to the camera viewpoint is visible. To be more precise, in this work we define overdraw as the fraction of fragments that are processed by the Fragment Processors and are finally occluded over the total amount of processed fragments. Overdraw is undesirable, as it represents useless activity, and an early and accurate visibility determination can significantly improve the performance and reduce the energy consumption. Ideally, an optimal system would draw a single opaque fragment per pixel (or screen position).

However, GPUs heavily suffer from overdraw which directly depends on the order in which fragments are processed. In a worst-case scenario, in which primitives arrive in a back-to-front order, a conventional Depth Test could not avoid the rendering of the occluded fragments. To provide an insight of the magnitude of this problem, Figure 2 shows the amount of overdraw for a set of modern games in a TBR architecture (Section 5 will detail the evaluation methodology). It can be observed that an average of 38% of the shaded fragments are eventually occluded, with some games such as Gravity (gra), Sniper3D (s3d) or Hot Wheels (hwl) reaching an overdraw factor around or over 50%.

In this paper we propose the Ω-Test[1], a novel micro-architectural technique that attacks overdraw and drastically reduces the amount of useless work performed by the Fragment Processors. Our approach relies on exploiting the frame-to-frame coherence (i.e., similarity between consecutive frames) [12], [13] by leveraging information from the Z values of the previous frame to speculatively detect which fragments will be occluded, instead of using only information from the current frame's Z-Buffer. Our technique does not introduce any error in the final rendered image since fragments that are wrongly identified as occluded are detected and eventually rasterized.

The main contributions of this work are the following:

- We propose a mechanism aimed at effectively reducing the overdraw factor within a scene, decreasing the number of fragments processed as well as the costly memory accesses to textures that they would require, hence improving the energy efficiency of the GPU while decreasing the execution time.
- We demonstrate that raw results of the Z-Buffer after rendering a frame are useful for the visibility determination of the next frame even after applying a coarsening factor of up to 16×16 to the data.
- We show how to integrate our technique in a TBR graphics pipeline. Experimental results, for a commercial set of applications, show that the Ω-Test achieves an average speedup of 16.3% and energy-delay (EDP) savings of 26.4% for the global GPU/Memory system.

The rest of the paper is organized as follows. Section 2 provides some background on the graphics pipeline of mobile GPUs, how the visibility problem is commonly solved, and reviews some other related works. Sections 3 and 4 describe the proposed Ω-Test and its implementation details. Section 5 describes our evaluation methodology whereas Section 6 quantifies and analyzes the achieved performance and the energy efficiency and, finally, Section 7 summarizes the main conclusions of the work.

2 Background and Related Work

2.1 Tile-Based Rendering Architectures

The architecture of modern GPUs can be categorized into two main families depending on how they process a scene: a) Immediate Mode Rendering (IMR), also known as full-framebuffer rendering; or b) Tile Based Rendering (TBR). While IMR processes and renders all the primitives on a per-frame basis, and it is the common design choice for high-end GPUs, TBR is aimed at improving the energy efficiency, and thus, it is commonly implemented in mobile GPUs. The key feature of TBR is that the screen area is divided in small regions of a fixed size called *tiles*. This partitioning is done in a way that allows the tiles to be individually rendered and benefits from the use of small and fast on-chip buffers for storing depth and output color values for a given tile. This dramatically reduces the amount of accesses to the main memory and the overall energy consumption of the system. Since our proposal targets mobile GPUs to further improve their energy efficiency, TBR is the baseline architecture we have chosen for this work.

Figure 3 shows the graphics pipeline of a TBR architecture, which is composed of two fundamental phases: the Geometry Pipeline and the Raster Pipeline. It can be seen that both phases are serialized, with the Tiling Engine acting as a mediator in between. This serialization is mandatory since the tile-based processing of

1. Named after the Z-Test but making an analogy with the Greek alphabet where Ω is the last letter, as Z is in the Latin alphabet.

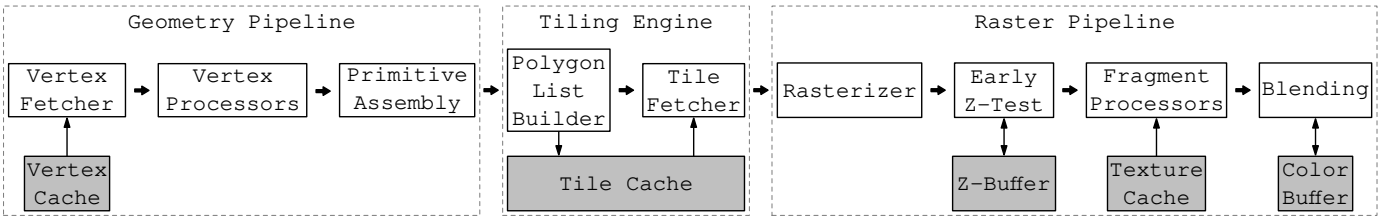


Fig. 3. Overview of the graphics pipeline for a TBR architecture, depicting the main stages that graphics workloads pass through.

TBR requires to process all the geometry first to determine which primitives belong to each tile. Only afterwards, the rasterization and rendering of the fragments can be done on a per-tile basis.

The Geometry Pipeline starts with a memory access stage to fetch the vertices of the scene. These vertices are transformed by geometric operations and assembled into primitives, typically triangles, which undergo a clipping process: primitives that are outside of the visible part of the scene (i.e., the part that the camera captures in its volume of vision, also known as *frustum view*) are removed and/or cut accordingly. Additional steps such as *backface culling* can also be applied to further reduce the number of primitives to be processed. Next, the Tiling Engine sorts the primitives into tiles, i.e., each tile contains a list with all the primitives that totally or partially fall inside the tile. These primitive lists (one per tile) are stored in main memory and are the input data to the Raster Pipeline.

The Tiling Engine is in charge of scheduling the tiles to be processed by the Raster Pipeline (also referred to as the Raster Unit). Note that multiple Raster Units can be used to process different tiles in parallel. The processing of a tile consists of several stages. First, the rasterizer tests the primitives at a pixel granularity to determine the pixels covered by them. If a pixel is covered by a primitive, the rasterizer interpolates the value of the primitive's attributes at the pixel's position.

Fragments are then grouped into groups of 2x2 adjacent fragments (a quad fragment) that are sent to the next stage of the pipeline, the *Early Z-Test*, a stage right before the Fragment Stage to avoid the shading of already-known occluded fragments. Recall that the Z-Buffer stores the depths of all the fragments processed so far. To determine if the current fragment is occluded, its depth is compared with that stored in the same position of the Z-Buffer. The resulting quad fragments are sent to the Fragment Stage, which contains the Fragment Processors. A Fragment Processor executes a *shader* program to compute the colors of each quad fragment which are stored in the Color Buffer. Finally, a Blending Unit allows for transparency effects by mixing the resulting colors with those already present in same Color Buffer position.

2.2 Background on Visibility Determination

As cited earlier, the most common approach to determine the visibility is the Depth Test, performed at fragment granularity. Modern GPUs typically implement this test in a stage called Early Z-Test by using a Z-Buffer that stores a value for each position of the visible area. Each of these values is usually the depth of the nearest fragment of that position. Thus, when a fragment is going to be processed, this stage checks whether it is closer to the camera than the fragment already present at the same position by comparing both depths. If the current fragment is farther (deeper) than the existing one in the Z-Buffer, it is discarded, avoiding costly shading and texturing. Otherwise, the current fragment's depth is kept in the Z-Buffer (overwriting the previous depth) which means that the current fragment is the closest one to the camera so far.

The Z-Buffer for a tile is built on the fly. Therefore, the Early Z-Test stage can effectively discard fragments when they arrive in a front-to-back order, i.e., later fragments that fall behind a closer one are discarded. However, the Early Z-Test is totally *ineffective* to avoid the shading of occluded fragments if they arrive in a back-to-front order. In any case, the major advantage of an Early Z-Test is that it always leads to the correct final image regardless of the order in which fragments arrived to the Fragment Processors. Its main drawback, on the other hand, is that its effectiveness is far from optimal since it leaves a significant amount of remaining overdraw, as it was shown in Figure 2 (average overdraw of 38%).

2.3 Related Work on Visibility Determination

Deferred Rendering. As opposed to traditional forward rendering schemes, in which the visibility determination and shading are performed on the fly as a whole, Deferred Rendering (also known as Deferred Shading) consists of determining the visibility of the whole scene before shading any fragment. Z-Prepass [14] can be seen as a basic Deferred Rendering approach since it decouples the geometry processing from the shading. In particular, Z-Prepass is a software technique able to eliminate overdraw caused by hidden surfaces that consists of two rendering passes at the application level. First, the entire geometry of the scene is calculated and rasterized with a *null* fragment shader, so that only the depth values are calculated and stored in the Z-Buffer. In the second pass, the depth values are in their final state, which allows the Early Z-Test to eliminate the overdraw of opaque surfaces.

G-Buffers [15], [16] are also used by applications (in the form of shader programs) to decouple the geometry processing from the shading, in addition to add more flexibility for doing lighting and material computations, as the G-Buffers can be used by the programmer for whatever needed.

A recent hardware-based Deferred Rendering approach is implemented in the PowerVR architecture [17] by adding a Hidden Surface Removal (HSR) stage in the traditional TBR pipeline that avoids performing two rendering passes at the application level. Instead, HSR iterates over all the fragments within a tile just calculating their position and depth to build a complete Z-Buffer before the actual shading pass is performed. This utilization of a Deferred Rendering scheme in a TBR pipeline is known as Tile-Based Deferred Rendering (TBDR). Recent academic work explored different alternatives of TBDR [8]. The main drawback of the best performing alternative is that it rasterizes all primitives twice, and so fragments are processed twice as well: once for calculating the depths in the visibility determination phase, and again for calculating the rest of attributes to continue down the pipeline. This forces designers to either increase the pressure over

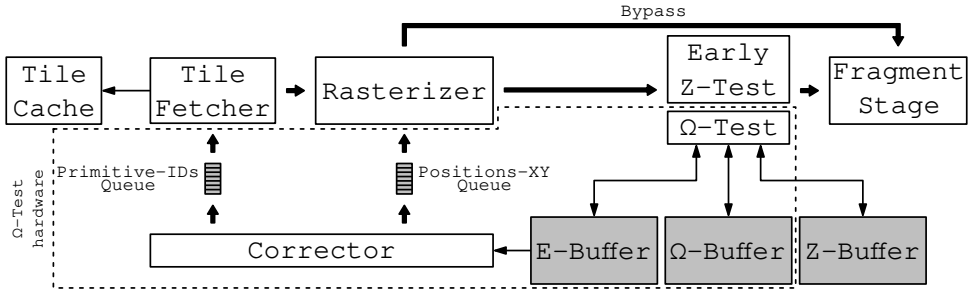


Fig. 4. Ω-Test implementation over a TBR architecture. The newly added structures and hardware are delimited by a dotted line.

the existing hardware (with the subsequent degradation of the execution time and energy consumption) or to include significant extra hardware to perform the extra computations of the HSR stage (duplicated rasterizer, Early Z-Test, and Z-Buffer) [6].

Contrarily, our proposal does not introduce timing overheads in the pipeline, and the added structures (Ω-Table, E-Buffer, correction queues; see Section 3) are small, on-chip buffers. As a reference comparison point, in [8] it is proposed Visibility Rendering Order (VRO), a technique that attacks overdraw by sorting objects in a front-to-back order, where authors quantitatively show that VRO outperforms TBDR. We have compared Ω-Test with VRO (see Section 6) showing that our approach beats VRO.

Another difference between Deferred Rendering (DR) and Ω-Test is that while the former requires the second pass to be done over the whole visible geometry, Ω-Test is highly more selective by only doing the correction phase (similar to DR's second pass) over a small amount of fragments whose visibility is mispredicted. I.e., only in the worst-case scenario, in which Ω-Test would fail all the visibility tests, meaning that all the fragments have to be sent to the correction phase for shading, Ω-Test would totally mimic the behavior of DR, since in the normal rendering phase (equivalent to DR's first pass) it would resolve the visibility of all the fragments while in the correction phase (equivalent to DR's second pass) it would perform the shading. As we will see in the Results Section, on average, only a small number of fragments have to be corrected, therefore outperforming a hardware-based DR such as TBDR.

HiZ occlusion culling. On the other hand, IMR GPUs suffer from a high memory BW utilization since the frame's Z-Buffer is stored in DRAM. To overcome this, HiZ (Hierarchical Z) occlusion culling was proposed [9], [18], [19], [20], [21] to coarsely cull primitives detected to be occluded in a region of the screen. This is implemented before the Early Z-Test (which operates at a fragment-level) and provides an earlier coarse Depth Test at a primitive-level. Therefore, HiZ culling avoids memory accesses to the frame's Z-Buffer by determining the visibility of primitive "chunks" rather than fragments. For that, a coarse-grained version of the frame's Z-Buffer is built on the fly. This buffer is called Hierarchical Z-Buffer and is computed in a per-region basis (given that the frame's Z-Buffer is subdivided into regions) for which a pair of (Z_{min}, Z_{max}) are calculated and stored. Consequently, primitives are sorted into these regions to perform the coarse occlusion culling test. This Hierarchical Z-Buffer can be further subdivided to get a coarser Z-Buffer, and hence the "hierarchical" term. The first implementation of a HiZ culling technique was proposed in [9], [18] as a software approach. This early proposal leveraged complex structures like oct-trees in order to efficiently determine the visibility of whole objects. Another HiZ implementation was proposed in [19] which implements a full hierarchy. In [20] a simpler HiZ was implemented with a single hierarchy level. In any case, HiZ does not replace the Early Z-Test stage. Whereas HiZ is mainly aimed at saving memory BW in IMR GPUs by saving costly accesses to the frame's Z-Buffer (in DRAM), a fragment-level Early Z-Test is still required to avoid the overdraw of occluded fragments. It is important to note that a HiZ is built on a per-frame basis, therefore it cannot eliminate as much overdraw if primitives arrive in a back-to-front order. Contrarily, Ω-Test further reduces overdraw by leveraging a *speculative* visibility determination since the Z-Buffer of the previous frame is used to predict the visibility of the current one.

Other recent works. More recent works leverage frame coherence to speculatively determine visibility. Visibility Rendering Order (VRO) [8] is a technique that sorts objects in a 3D scene based on the front-to-back order from the preceding frame. Another recent technique is Early Visibility Resolution (EVR) [10] which uses the farthest point for each tile in a frame to predict occluded primitives in the next frame, with the aim of processing those presumably occluded primitives as the final ones. Both VRO and EVR use information from the preceding frame to re-sort the order in which objects/primitives are processed in the current frame to increase the effectiveness of the Early Depth Test. VRO solves the visibility problem at the *object* level while EVR solves it at the *primitive* level. Differently, our Ω-Test operates at the much finer granularity of *fragments*, being able to overcome not only inter-object overdraw but also intra-object overdraw, therefore, capable of outperforming VRO as shown in Figure 12.

3 The Ω-Test Approach

The proposed Ω-Test slightly modifies the behavior of the Early Z-Test stage, where the visibility of fragments is determined. After performing the original Early Z-Test, and updating the Z-Buffer if necessary, a second test is performed but this time using an Ω-Buffer, a new structure similar to the Z-Buffer which holds the Z values corresponding to a given tile of the previous frame. If the Ω-Test is passed, the fragment can proceed to shading. Otherwise, the fragment is discarded since it is assumed it will be again occluded in the current frame (as it was indeed occluded in the previous frame according to the contents of the Ω-Buffer which corresponds to the Z-Buffer of the previous frame). Note that, unlike the Z-Buffer, which may be updated whenever a fragment passes the Early Z-Test, the Ω-Buffer is never updated regardless of the Ω-Test outcome. There is no need for such updates because it is the Z-Buffer the one in charge of keeping the most up-to-date Z values for the current frame.

Figure 4 shows the modifications made to the baseline TBR architecture to implement our proposal (delimited by a dotted

line to better differentiate them). Our technique reduces overdraw with respect to TBR since each fragment has to pass a second test. This Ω-Test can be seen as a *backup* test for the cases when the traditional Early Z-Test does not work efficiently (e.g., when primitives do not arrive in a back-to-front order): we still have a second resort to discard a potentially occluded fragment by using the Z value from the previous frame. Due to frame-to-frame coherence, if a fragment was occluded in the previous frame, it is highly likely that it will also be occluded in the current frame. However, this approach does not guarantee that fragments discarded by the Ω-Test will not be visible in the final image. It may happen that a fragment that does not pass the Ω-Test is finally visible (e.g., an object that suddenly appears in front of other objects) leading to a potential *error* in the tile. For such cases, our approach implements a simple error detection and correction mechanism right after the tile is shaded, which will be further detailed in Section 4.

To be more specific, three situations may happen regarding both the Z-Test and the Ω-Test outcomes:

- Case 1: Early Z-Test not passed (regardless of the Ω-Test result). The fragment is safely discarded as the decision is based on information from the current frame (Z-Buffer) and no speculation is done (no errors are generated).
- Case 2: Early Z-Test passed, Ω-Test not passed. The fragment is speculatively discarded as the decision is based on information from the previous frame (Ω-Buffer). This is the only case where a potential error might be generated.
- Case 3: Early Z-Test passed, Ω-Test passed. The fragment is sent to shading. Still, this could produce a false positive test which may lead to overdraw if the fragment is eventually occluded.

A main characteristic of a TBR graphics pipeline is that the working unit is a tile. However, after finishing the processing of a tile, the valuable information contained in the Z-Buffer is discarded. As we want those depth values to be used in the next frame, the Z-Buffer must be preserved somehow. We employ a frame-level structure called Ω-Table to store the information of the Z values of all the tiles from the previous frame. If we had decided to implement this structure in on-chip buffers, the storage needs for a frame in Full-HD resolution (1920x1080 pixels) would be around 8 MBytes, which would contradict the TBR philosophy that encourages the use of small (on-chip) memories for a better energy efficiency.

A second solution consists of storing the Ω-Table in DRAM. The main drawback of such an approach is the intensive use of main memory because of the extra transfers needed before and after processing each tile, resulting in prohibitive energy costs (recall that DRAM consumes more than 50% of the baseline GPU's energy, as reported in Section 1). We quantitatively evaluated this solution of storing the Ω-Table in DRAM. Unfortunately, the net effect in the overall system energy consumption was negative, since the additional DRAM accesses more than offset the benefits coming from the overdraw reduction, so using DRAM for fully storing the Ω-Table was also discarded.

To efficiently cope with the storage needs associated to our approach while not incurring in significant energy costs, our final solution consists of not storing all the Z values from the previous frame but a small set of representative ones. Even though this results in a loss of information, as we will describe next, we observed that just keeping a few representative values per tile was as efficient as keeping the complete tile's Z-Buffer. Obviously, there is a small number of induced errors for not using precise information but the mechanism for detecting and correcting errors (see Section 4.3 for additional details) fixes them. In addition, the incurred overhead from the correction phase pays off with respect to the saved energy from neither using large on-chip memories nor relying on DRAM for storing the previous frame's Z values. One alternative approach we have not evaluated is a hybrid implementation where the Z values are stored in DRAM while an on-chip Ω-Table is still used as a cache for them to support much higher display resolutions.

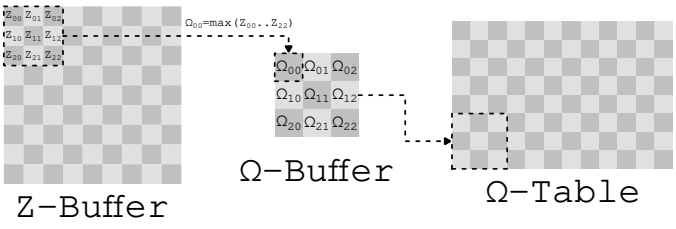


Fig. 5. Compression scheme example for a 3x3 coarsening factor over a 9x9 pixel tile. The Z-Buffer is processed in 3x3 pixel areas as determined by the coarsening factor. For each area, the maximum is computed and stored onto the Ω-Buffer. When all the 3x3 areas are processed, the Ω-Buffer is flushed onto its corresponding position in the Ω-Table.

There are multiple ways of selecting a set of representative values to compress the Ω-Table. However, we avoided traditional compression algorithms because of their hardware complexity and energy cost. As commented before, the Ω-Test already generates some errors (that are later corrected) due to the fact it relies on depths from the previous frame. Therefore, there is no need to be 100% precise with the information to be used, since our approach can afford some extra initial errors. I.e., a fast and simple scheme that loses some information can be more appropriate for our purposes than a complex lossless compression scheme. As a trade-off solution, we decided to make use of *coarsening* and conventional *aggregate functions* (e.g., maximum, minimum, arithmetic mean) to select the set of representative Z values. The idea of coarsening is to keep a single value for a set of neighbor pixels based on the observation that neighbor pixels tend to have the same or very similar depths. On the other hand, aggregate functions are simple to implement in hardware.

We define a *coarsening factor* that represents the granularity level we use to build the Ω-Table. For example, assuming tiles of 9x9 pixels, a 3x3 coarsening factor will break the tile down into 9 non-overlapping squares of 3x3 pixels. Then, each of these 3x3 squares will go through the aggregate function. The result will be a matrix of 3x3 elements containing the resulting values of the applied aggregate function. Summarizing, we go from a 9x9 matrix down to a 3x3 one, which reduces the storage needs by a factor of 9x. To better illustrate this, Figure 5 depicts this compression scheme based on using a coarsening factor followed by an aggregate function.

As a summary of the storage needs, Table 1 shows the memory required by the Ω-Table depending on the coarsening factor for a common HD screen resolution (1280x720 pixels) and 16x16 pixel tiles. The 1x1 coarsening factor means that all the tile pixels are stored in the Ω-Table, i.e., equivalent to not applying coarsening. The 16x16 coarsening factor is the maximum possible for the assumed tile size (16x16 pixels) and means that the whole tile is represented by a single pixel. To better understand how the

TABLE 1
Ω-Table storage needs for some coarsening factors that are possible in a 16x16 pixel tile and assuming a 1280x720 screen resolution.

Coarsening factor	Ω-Table size
1x1 (or no coarsening)	3.52 MiB
2x2	900 KiB
4x4	225 KiB
8x8	56.25 KiB
16x16	14.06 KiB

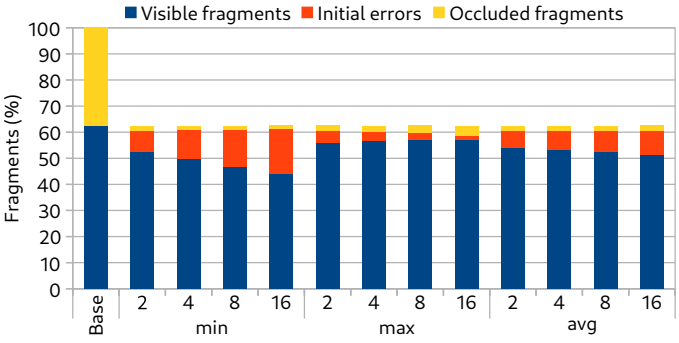


Fig. 6. Effect of different coarsening factors (from 2x2 to 16x16) for 3 aggregate functions (min, max, average), showing the fraction of fragments (normalized to the baseline) that are either visible, appear as initial errors (that have to be corrected), or are occluded. Each bar corresponds to the average of the 10 evaluated benchmarks.

Ω-Table size is calculated, let us consider the case of an 8x8 coarsening factor. In this case, only 4 values (2x2 coarse pixels of 8x8) will be stored. Given that a 1280x720 screen has 3600 tiles (80x45) and provisioning 32 bits for each Z value, the final size of the Ω-Table will be 56.25 KiB (4 values $\times$ 3600 tiles $\times$ 4 bytes) resulting in a storage reduction of 64x with respect to not using coarsening (the 1x1 case in Table 1).

Next, we have measured the amount of initial errors as a result of using different coarsening factors along with different aggregate functions (maximum, minimum and average). Figure 6 shows the fraction of errors with respect to the total amount of processed fragments (including also the shading and the residual overdraw) where each bar represents the average of all the evaluated benchmarks. Let us analyze first how the different aggregate functions behave. The *minimum* function keeps the depth of the fragments closer to the camera, so the Ω-Test is more restrictive and ends up discarding more fragments than needed. Overdraw is reduced at the cost of generating too many errors, as it can be seen in Figure 6. On the other hand, using the *maximum* function makes the Ω-Test more permissive since we compare against the deepest Z of the group. Overdraw is not reduced as much as with minimum but the number of errors is highly reduced. Finally, using the *average* as the aggregate function leads to a trade-off between errors and overdraw. As our goal is to generate as few errors as possible because of the overhead of the correction phase, we have chosen the maximum as the aggregate function for the Ω-Table, i.e., the Z values of a group will be represented by the most distant one to the camera.

Regarding the coarsening factor, Figure 6 also shows that the biggest possible coarsening factor of 16x16, when combined with the maximum aggregate function, does not incur a significant potential loss (less than 3% of errors). As a result, for the final design of the Ω-Table we have chosen to use a coarsening factor of 16x16 which in practice means that each tile will be represented by only one Z value, in particular the maximum one. This results in a reduction of the storage needs by a factor of 256x without significantly penalizing the potential. As for the storage needs, when using a 16x16 coarsening factor, the size of the Ω-Table lowers to about 14 KiB (Table 1) which can be easily allocated as a small on-chip buffer.

In any case, although the use of coarsening leads to a very good compression ratio, other schemes aimed at reducing the storage needs with a potentially smaller accuracy degradation will be considered for future work, such as precision reduction, quantization and downsampling.

4 Efficient Management of Errors

4.1 Scenarios that may Lead to Potential Errors

The main advantage of the Ω-Test is that it decreases the number of quad fragments that are executed in the Fragment Processors by means of *speculatively* discarding the occluded ones based on the contents of the Ω-Table. The downside is that it may lead to some errors. To better illustrate these scenarios, Figure 7 shows the possible cases that could produce potential errors and how Ω-Test handles them. Figure 7-(a) shows the initial scene (frame i) with two overlapping primitives rendered in back-to-front order. The first of the analyzed cases (depicted in Figure 7-b) illustrates the case of a primitive moving away from the camera. Recall that each fragment of a primitive has a Z' value from the previous frame (retrieved from the Ω-Table) in addition to its current Z value. In this particular case, as primitive B has moved backwards, their fragments will fail the Ω-Test ($Z > Z'$).

Figure 7-(c) illustrates the case of using a δ margin, a mechanism aimed at mitigating errors that will be further explained in Section 4.2, where a big amount of the errors can be reduced provided that δ is large enough to *mask* such backward movement. Fragments from primitive B will now pass the Ω-Test ($Z < Z' + \delta$) whereas the ones from primitive A that were occluded in the previous frame will still fail.

Figure 7-(d) shows the opposite movement, i.e., a primitive that comes closer to the camera (or conversely, when the camera moves forward making the primitive to become closer). This case does not generate errors since the current Z values are closer than those from the previous frame ($Z < Z'$) and the Ω-Test will pass as intended. At the most, this case could generate overdraw. As an example, our benchmarks include racing games such as Beach Buggy Racing or Hot Wheels (refer to Table 3) where forward camera movements are usual.

Another case that may lead to potential errors corresponds to the lateral movement of a primitive, as in Figure 7-(e). In this example, as primitive B moves from left to right, it hides the right-side part of primitive A while unveiling its left-side part. In this case, some fragments from primitive A that were occluded in frame i become visible in frame $i+1$, leading to potential errors in the final image.

As a final case, Figure 7-(f) shows how the use of the maximum aggregate function with the coarsening scheme considerably reduces those lateral-movement-based errors in detriment of overdraw. In this case, since all the fragments from primitive B will pass the Ω-Test, no errors will be generated. However, as the two primitives are rendered in back-to-front order, it will generate overdraw as in the baseline.

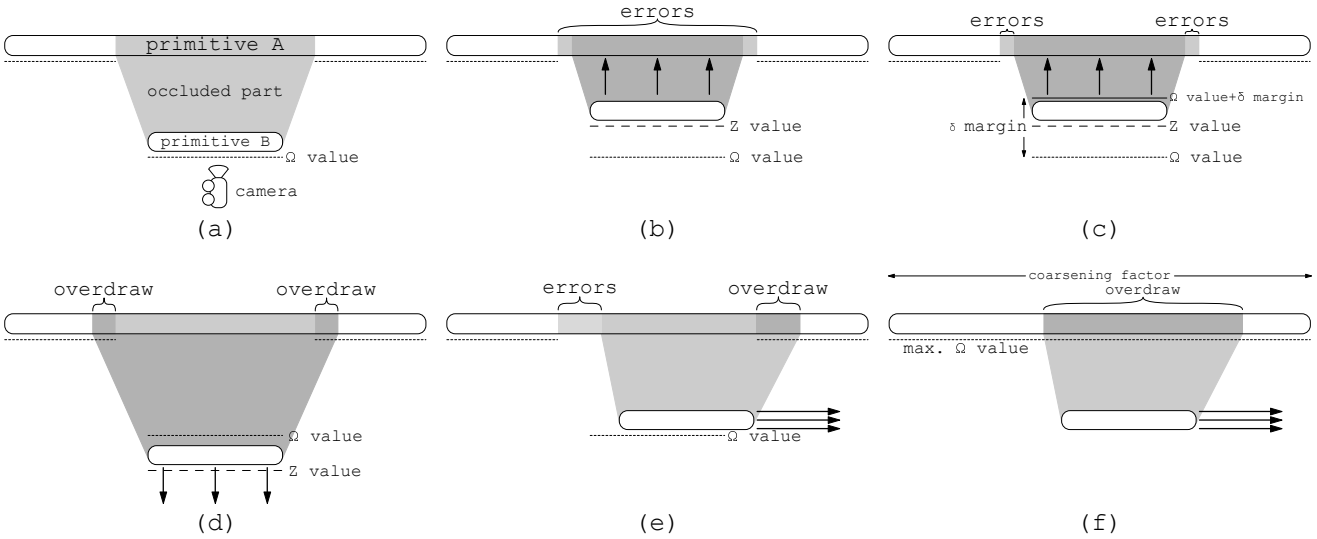


Fig. 7. Set of scenarios that could produce potential errors in Omega-Test. (a) Initial scene (frame i) with two overlapping primitives rendered in back-to-front order. (b)-(f) depict the frame i+1 for a number of possible movements of primitive B. Cases (b) and (c) shows the primitive moving away from the camera which lead to errors as depicted. In case (b) no δ margin is used, thus producing errors in the area that primitive B overlaps with A. In case (c) a δ margin is used and errors are mitigated, only remaining those in the border of primitive A. Case (d) shows primitive B coming closer to the camera. This case does not generate errors as current Z values are nearer than the Ω values, thus passing both tests. In cases (e) and (f) primitive B moves laterally (panning). In case (e) no coarsening is used, producing errors on the area that primitive B unveils from A and it leaves overdraw on the area it moves to. In case (f) a coarsening with the maximum aggregate function is applied which avoids the errors in (e) since the Ω values tend to be farther than the Z values (thanks to using the maximum function).

Finally, note that any error, either coming from lateral movements or as a result of using an "aggregated" Z value to represent a whole tile, is solved by adding a final correction step as described in Section 4.3.

4.2 Mitigating Errors: the Delta (δ) Margin

It is very common to have scenes in which the Z-Buffer of many tiles remains constant across consecutive frames. In this case, our technique acts ideally because it does not produce errors nor draws unnecessary fragments. However, as described previously, there are other scenes in which the objects or the camera move, which may affect a significant amount of (or all) the tiles in the frame. As seen in the cases depicted in Figure 7-(b) and Figure 7-(c), a particular type of movement that potentially lead to errors happens when objects move away from the camera.

To be able to tolerate such movements while reducing the amount of errors, we include a small safety margin for the Ω-Table called delta (δ) margin. By doing this, we relax the Ω-Test condition so that it is equivalent to slightly moving the depths of all the fragments a bit farther. The rationale behind this δ margin is to be more permissive by not eliminating fragments that belong to primitives that have slightly moved away from one frame to the next. By using this safety margin, Ω-Test becomes more flexible and incurs less errors. On the other hand, the amount of overdraw is not significantly hurt because this δ margin is very small.

In essence, we are trading errors for overdraw, i.e., the δ margin helps reduce the amount of errors at the expense of not being able to avoid some overdraw. To better understand the effect of using this safety margin, we have analyzed a wide range of δ values and measured how the amount of errors and the overdraw factor are affected. It can be observed in Figure 8 that as we increase δ, more fragments pass the Ω-Test, resulting in higher overdraw. However, the number of errors is reduced. Contrarily, smaller δ values are less tolerant to depth changes in the Ω-Buffer, causing more errors but being more effective at reducing overdraw. Figure 8 shows that the best δ is 0.0005 for many games (recall Z values are normalized in the range [0,1] where 0 corresponds to the near plane and 1 represents the far plane). Note also that frame-to-frame coherence has a significant influence on δ, because the more coherence the smoother the movements will be and, therefore, smaller δ values will suffice.

However, using a static δ for all the games does not provide the best trade-off, given the high variability that can be found in games. To cope with this inter-frame variability, we have implemented a very simple dynamic scheme that defines a δ value for each frame that is adapted depending on whether the objects within a frame move away or not.

The adaptive technique changes δ based on frame-level overdraw/error ratios. We define a cost function (Equation 1) whose inputs are the number of overshaded fragments and errors. In (1) c_o is the relative cost associated to overdraw and c_e is the relative cost associated to errors, always speaking in terms of energy. Experimentally, we have quantified the cost of correcting an error to be about 3x higher than shading a fragment, so the adaptive scheme prioritizes reducing the amount of induced errors with weights $c_o = 0.25$ and $c_e = 0.75$. Finally, o and e represent the per-frame amount of overdraw and errors, respectively.

$$cost(o,e) = c_o \times o + c_e \times e \tag{1}$$

The dynamic δ scheme uses a table of eight δ values (0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5); an index pointing to the current δ, a variable that indicates in which direction we move this index, and two global counters to account for the number of errors and overdraw for the whole frame. The initial value for δ is set to 0.0005 (since it was the best static δ). The adaptive scheme operates as follows. When finishing rendering a frame, the cost function (1) is evaluated and compared with that of the previous frame (held in a global register). If the current cost is higher, we

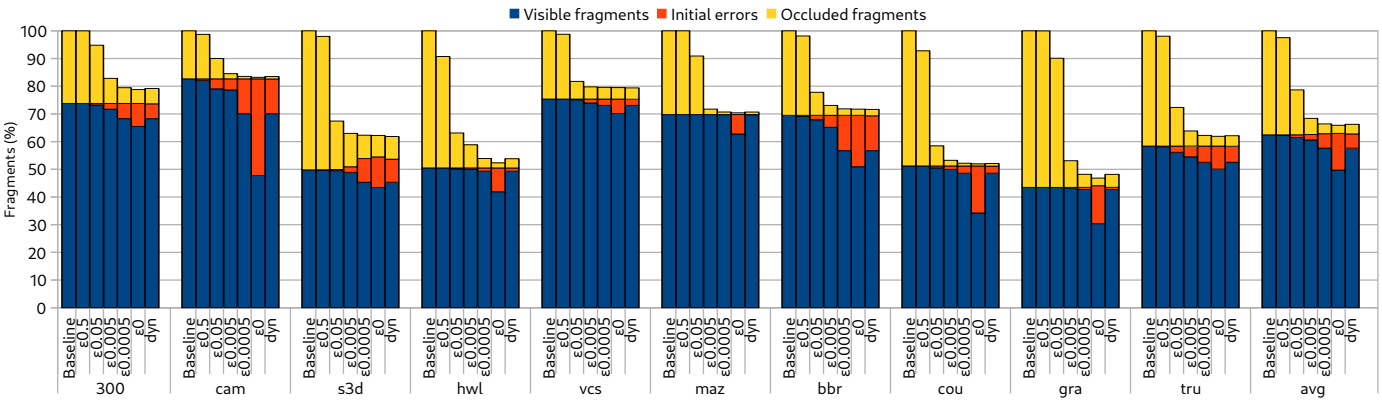


Fig. 8. Study of the ratio overdraw (occluded fragments) vs. initial errors for several static δ margins (0.5, 0.05, 0.005, 0.0005 and 0) normalized to the baseline. The last bar of each benchmark (dyn) corresponds to the dynamic δ implementation.

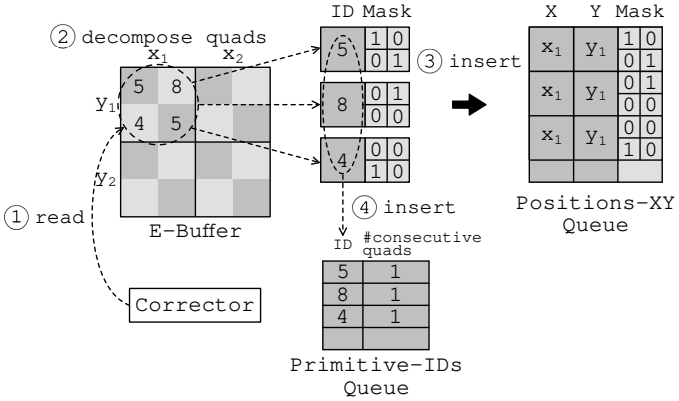


Fig. 9. Correction phase. A number other than -1 in any location of the E-Buffer indicates an error to be corrected. In this example there are 4 errors in the top-left quad, corresponding to primitives 5, 8 and 4.

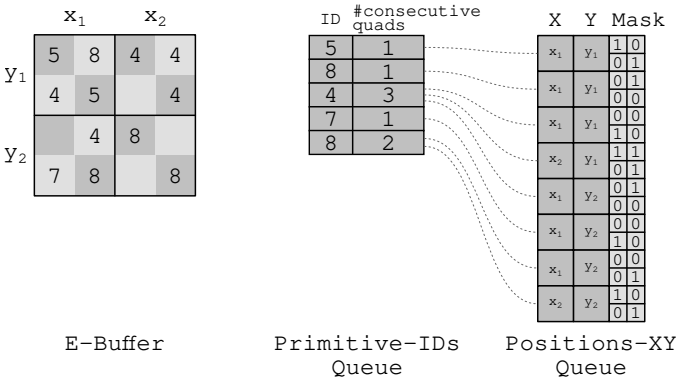


Fig. 10. Example of the final state of the queues given an E-Buffer of 4x4 pixels (4 quads).

change the direction of the index of the δ table, otherwise, we move the index in the direction shown by the cited variable. If the table limit is reached, the index saturates.

The dynamic δ scheme is able to effectively reduce the fraction of errors as intended. In particular, it can be observed in Figure 8 that the average fraction of errors (over the total amount of fragments) is just 5.08% thanks to using the dynamic δ scheme.

Another approach we have evaluated is a *per-tile* δ margin. However, we have not included this more refined mechanism in the final implementation due to the storage overhead and the poor benefit obtained. As each tile needs to store its own δ, 32KiB are required (assuming FullHD resolution, 16x16 tiles, and 4 bytes per δ). The experimental results showed that the initial errors were reduced, on average, from 5.08% to 5.02% making it not worthy.

4.3 Error Detection and Correction

As the Ω-Test might lead to discarding a fragment which is visible in the final image, we need a mechanism to detect and correct these errors. Errors are generated in the Early Z-Test stage where it is checked whether a fragment must proceed or not to the Fragment Processors for shading. A fragment that passes the Early Z-Test but not the Ω-Test (case 2 explained in Section 3) could be a potential error. However, note that this fragment can be *hidden* by another visible fragment rendered on top of it. In this case, the Ω-Test has avoided an undesired overdraw case, saving useless work. However, if no fragment is ever written in that position of the tile, a *gap* would be left in the final Color Buffer. Obviously, these induced gaps cannot be propagated to the Frame Buffer and a corrective action must take place.

To keep track of the potential errors, some additional data structures are needed. In particular, we use a two-dimensional array called E-Buffer, with the same dimensions of a tile, where each element is associated to a pixel and stores a primitive's *identifier*. Once the Early Z-Test is passed, we perform the Ω-Test. If the Ω-Test fails, we store the primitive ID in its corresponding position in the E-Buffer which indicates that this primitive *could* potentially cause an error (or gap) in that position. Otherwise, if the Ω-Test succeeds, we store in the E-Buffer a special ID (a -1 in our case). Therefore, a final -1 in any position of the E-Buffer indicates that there is no error (or gap) left in that position.

When all opaque primitives have been rasterized and the Early Z-Test has no more fragments to check, the E-Buffer is in a final state which identifies where the final errors (non-rendered fragments) are located. At this point, the correction phase is ready to be performed. One important thing to note is that this phase is only triggered if an error is found in the E-Buffer, thus it is not unnecessarily launched when there are no errors. To quickly verify this, a global counter is used which is increased when a primitive ID is stored in the E-Buffer (overwriting a -1); and it is decreased when a -1 is stored (overwriting a valid primitive ID).

To better illustrate the process, Figure 9 shows how the Corrector works. First, it reads quad fragments from the E-Buffer

(step ①). In this example, the top-left quad contains 4 errors corresponding to 3 different primitives (IDs 5, 8, 4). For each *different* primitive within the quad, a 4-bit (2x2) visibility mask is generated whose active bits are the pixels that the primitive occupies inside the quad (step ②). Thus, a quad can generate up to four visibility masks in the case there are four errors with four different Primitive IDs (this is the worst case). These masks along with the quad position are inserted into the Positions-XY queue (step ③). If all the valid primitives of a quad (according to the visibility mask) are equal (this is the best case and, fortunately, the most common) another global counter is incremented, indicating the number of consecutive quads (in scan-line order) from the same primitive. When a new primitive is found, an entry is inserted into the Primitive-IDs queue (step ④), containing both the ID of the previous primitive and the value of the global counter (indicating the number of consecutive quads from the same primitive) which is set to zero again.

To better understand how the Corrector works, Figure 10 depicts an example with an E-Buffer containing different errors. The entries in the E-Buffer with a number (a primitive ID) correspond to positions where there is an error for that particular primitive, whereas empty cells represent a correct pixel (i.e., a -1). For the sake of visibility, we assume 4x4 pixel tiles. The final state of the Primitive-ID queue and the Positions-XY queue for this example is also shown in Figure 10.

As soon as there is a primitive in the Primitive-ID queue, the Tile Fetcher starts working on the corrections. Under this correction mode, the Tile Fetcher rather than querying the Tile Cache for a new primitive, gets it from the Primitive-ID queue. Additionally, the number of quads of the same primitive is provided by this queue to tell the Rasterizer how many errors will be corrected. This mechanism avoids, on the one hand, that the Tile Fetcher reads the same primitive multiple times when correcting several quad-fragments from the same primitive and, on the other hand, to fill the Primitive-ID queue with redundant data.

Under the correction mode, the Rasterizer has a slightly different behavior. In particular, only the fragments to be corrected are generated for a given primitive. That is, if a triangle only has one erroneous pixel, this is the only fragment that will be generated. To do that, the Rasterizer calculates the barycentric coordinates of the first fragment and the X and Y increments, as usual. Note that the (X,Y) coordinates where the error is located are obtained from the Positions-XY queue, along with the visibility mask (refer to Figure 9). Then, the quad fragment is sent to the Fragment Processors for a proper shading. Note that the quads submitted for correction do not undergo the Early Z-Test because they are known to be visible.

After the correction phase, the graphics pipeline continues working as usual. When the tile is completely rendered and the Color Buffer is computed and flushed, the pipeline is ready to start with a new tile.

Finally, there is a challenging situation that happens when the Tile Fetcher finds a *transparent* primitive. A primitive is considered transparent if its blending attribute is active, meaning that all the fragments from this primitive have to mix their rendered colors with the existing ones in the Color Buffer. The problem is that when the transparent fragment is processed, the Color Buffer must contain the color of the previously generated opaque fragment, and if it has been erroneously filtered out by the Ω-Test, it would be erroneously mixed with a black fragment. To overcome this situation, the correction phase is triggered as soon as a transparent fragment arrives in the Early Z-Test stage, to make sure that any potential error in the opaque geometry is corrected before processing the transparent fragment. When the correction of the errors for opaque primitives is done, the pipeline can continue processing the transparent fragment and the normal operation of the Tile Fetcher is resumed. This causes a stall in the pipeline that might hurt the performance. Fortunately, such interleaving pattern is not usual, as it does not comply with the OpenGL recommendation about rendering order which states that transparent primitives must be rendered after opaque ones to produce the correct output image [22]. In particular, out of the 10 evaluated benchmarks, only Maze 3D (maz) presents this uncommon interleaving pattern with a small number of primitives. As expected, Ω-Test produces the same output as the baseline GPU, and due to the low occurrence of such interleaving pattern, the performance of Maze 3D (maz) is not degraded at all as it can be seen in Figure 13.

5 Evaluation Methodology

5.1 Simulator Infrastructure

We have used TEAPOT [23], a simulation framework that includes a cycle-accurate simulator for GPUs, including models based on Mali's Utgard architecture [24]. TEAPOT includes timing and power models based on well-known tools: McPAT [25] for power estimation, and DRAMSim2 [26] for modelling DRAM and the memory controllers. The benchmarks have been run either in a real smartphone or in an Android Virtual Device (AVD) [27] to obtain a trace of OpenGL commands. The OpenGL [28] traces have been obtained with GAPID [29], a graphics debugger that allows to inspect the graphics commands of animated applications. In particular, the OpenGL trace is executed with the GAPID replay tool (`gapir`) over an instrumented Gallium Softpipe Driver [30] to obtain the final trace. This trace is consumed by the cycle-accurate simulator, which produces timing reports and a file of activity factors. Those activity factors are employed by the power model to generate a power report.

Table 2 shows the GPU simulation parameters, resembling the ARM Mali-450 GPU that we have used to evaluate our proposal. This Table also shows the configuration parameters of the structures used by the Ω-Test (namely, Ω-Table, E-Buffer, Primitive-ID queue and Positions-XY queue). All these structures have been modelled and included in the timing and power model of the GPU. In particular, their area overhead has been measured to be 2.29% of the total area of the GPU. As mentioned in Section 3, the biggest structure is the Ω-Table (with a size of 14.06 KiB thanks to the 16x16 coarsening factor) which corresponds to a relative area of 1.70% of that of the GPU.

5.2 Benchmarks

Table 3 shows the set of benchmarks we have used in our experimental evaluations. We employ commercial applications, which do not require any modification at the software level to benefit from the Ω-Test. The games have been selected based on their popularity in number of downloads on the Google Play Store. Note that we have only considered 3D games since our technique does not apply to 2D games. Figure 11 shows a single frame for eight of the evaluated benchmarks to provide an insight of the complexity present on their scenes.

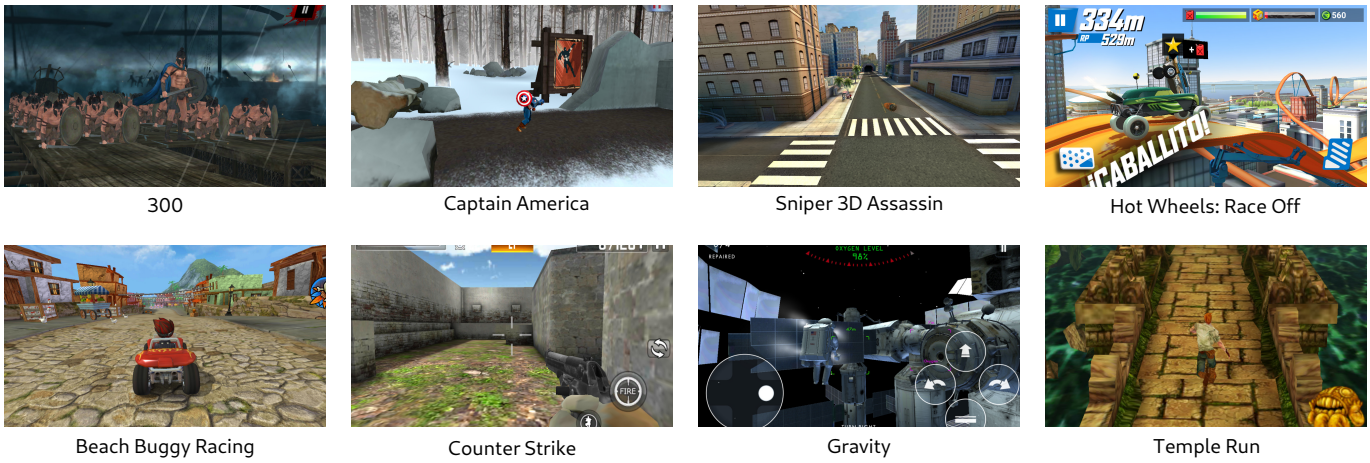


Fig. 11. Images of the evaluated benchmarks showing the complexity of the scenes.

TABLE 2
GPU Simulation Parameters.

Baseline GPU Parameters	
Frequency	600 MHz
Voltage	1.0 V
Scale Integration	22 nm
Screen Resolution	1280x720
Tile Size	16x16 pixels
Main Memory	
Frequency	400 MHz
Voltage	1.5 V
Latency	50-100 cycles
Bandwidth	4 B/cycle (dual channel LPDDR3)
Size	1 GiB
Queues	
Vertex (Input & Output)	16 entries, 136 bytes/entry
Triangle & Tile	16 entries, 388 bytes/entry
Fragment	64 entries, 233 bytes/entry
Color	64 entries, 24 bytes/entry
Caches All of 64 bytes/line, 2-way associativity	
Vertex Cache	4 KiB, 1 bank, 1 cycle
Texture Caches (x4)	8 KiB, 1 bank, 1 cycle
Tile Cache	128 KiB, 8 banks, 1 cycle
L2 Cache	256 KiB, 8 banks, 2 cycles
Color Buffer	1 KiB, 1 bank, 1 cycle
Depth Buffer	1 KiB, 1 bank, 1 cycle
Non-programmable stages	
Primitive assembly	1 triangle/cycle
Rasterizer	1 attributes/cycle
Early Z test	8 in-flight quad-fragments
Programmable stages	
Vertex Processor	4 vertex processor
Fragment Processor	4 fragment processors
Ω-Test hardware	
Ω-Table	14 KiB
Positions-XY Queue	64 entries, 13 bytes/entry
Primitive-ID Queue	64 entries, 8 bytes/entry
E-Buffer	1 KiB
Corrector	4 quad-fragments/cycle

TABLE 3
Evaluated benchmark set.

Benchmark	Alias	Description	Downloads (M)
300	300	Hack & slash	10-50
Captain America	cam	Beat'em up	1-5
Sniper 3D Assassin	s3d	Shooter	100-500
Hot Wheels: Race Off	hwl	Racing	50-100
Vegas Crime Simulator	vcs	Sandbox & Crime	100-500
Maze 3D	maz	Labyrinth	10-50
Beach Buggy Racing	bbr	Racing	50-100
Counter Strike	cou	Shooter	10-50
Gravity	gra	Action	1-5
Temple Run	tru	Adventure arcade	100-500

6 Experimental Results

We have evaluated both a baseline GPU design and a GPU that implements the Ω-Test with a coarsening factor of 16x16, the maximum function for "aggregation", and with the dynamic δ mechanism for mitigating errors.

6.1 Overdraw Reduction

Figure 12 shows a first set of results for the Ω-Test approach. In particular, it reports a breakdown of all the rendered fragments for each benchmark, differentiating the fraction of occluded fragments (overdraw –in yellow–), transparent fragments, the induced errors by the Ω-Test (in red), and the visible fragments (in dark blue). As our technique is aimed at optimizing the visibility problem, it can only attack the overdraw fraction. Note that an ideal visibility determination approach would remove this fraction completely. For comparison purposes, we have evaluated VRO [8], an advanced Hidden Surface Removal (HSR) technique that sorts objects front-to-back based on frame-to-fame coherence (see Section 2.3 for further details). On average, Ω-Test reduces the baseline's overdraw by 32.7%, still leaving a residual overdraw of 4.5% that is unable to eliminate. These initial results show that our proposal is pretty close to the ideal case, differently to VRO that leaves an average residual overdraw of 22.2%. Although both VRO and Ω-Test leverage frame-to-frame coherence to reduce overdraw, they achieve their goals using quite different strategies. VRO works at a command granularity by sorting commands so that they are processed in a front-to-back order. Ω-Test, however, operates at a much finer granularity (fragment level) which is more effective since it can avoid the useless shading of occluded fragments in primitives of the same object (intra-object overdraw) or from partially overlapping objects. It is worth noting that intra-object overdraw is significant in the evaluated benchmarks and in mobile games in general since they typically include complex objects in a single draw call. For instance, in Hot Wheels (hwl), the whole

city in the background (made of multiple and complex buildings - see Figure 11) is a single object rendered in a single draw call. As expected, there is a lot of intra-object overdraw that VRO cannot eliminate whereas the proposed Ω-Test can indeed.

On the other hand, errors (i.e., wrongly discarded fragments that must be indeed shaded) may hurt performance since they have to be fixed on the correction phase, so it is desirable to reduce them as much as possible. On this regard, our proposal generates an average of 5.1% *initial* errors that are fixed on the correction phase. As a side note, the average number of errors to be fixed per tile is, in absolute terms, just 13 out of 256 pixels in a 16x16 tile.

6.2 Speedup and Memory Utilization

Now, let us see the overall net impact on the execution time due to both the overdraw reduction and the errors that have been fixed. Figure 13 shows the speedup achieved when the Ω-Test is included in the graphics pipeline. It achieves an average speedup of 16.3%, with a maximum speedup of 32.7% for Gravity (gra). Figure 13 also plots a configuration with perfect visibility knowledge that shows an average upper bound of 17.9% for the speedup; and the aforementioned VRO which achieves an average speedup of 12.75%. As expected, Ω-Test is pretty close to the perfect scenario because of the low residual overdraw it leaves (a mere 4.5%) and beats VRO. Focusing on the realistic Ω-Test, although it greatly reduces overdraw in many applications, in some cases the execution time is not reduced in the same proportion. This is because of the overhead incurred on correcting errors and also due to the pipeline stall to wait for that correction phase. To provide an insight on this, it is necessary to look at the accesses that end up going to DRAM as a result of misses in the Texture Cache (the equivalent to an L1 in a CPU). As expected, applications with small textures are more likely to obtain a higher hit rate in the Texture Cache, therefore, reducing their overdraw does not save many DRAM accesses, and so we are not saving as much latency. Contrarily, applications with detailed textures exhibit a lower hit rate in the Texture Cache and go more frequently to DRAM. As such, reducing the overdraw factor in these games results in a more significant impact. This is the case of Gravity (gra), Hot Wheels (hwl) and Beach Buggy Racing (bbr).

To provide a better insight of this effect, Figures 14 and 15 show the accesses to the Texture Cache, the L2 and DRAM. While Figure 14 breaks down the accesses to the Texture Cache in hits and misses (the latter end up going to L2), Figure 15 shows how many of the memory requests are served by DRAM. Let us focus on two representative examples: Counter Strike (cou) and Gravity (gra). First, recall from Figure 12 that they have a similar net overdraw reduction of 47.9% and 51.8%, respectively. However, these overdraw reductions are translated into respective speedups of 17.4% and 32.7%. If we now look at their memory behavior (Figure 14) we observe that, thanks to using the Ω-Test, there is a reduction in their overall number of memory accesses (46.2% and 41.2%, respectively) which correlates with the reported overdraw reduction. However, because of their different texture complexity, while Gravity reduces its DRAM accesses by 30.3%, Counter Strike barely reduces them (a mere 3%) as shown in Figure 15.

6.3 Energy Savings

Next, let us focus on the energy savings achieved by the Ω-Test and reported in Figure 16. It can be seen that our approach provides average energy savings of 15.17% (and up to 26.9%

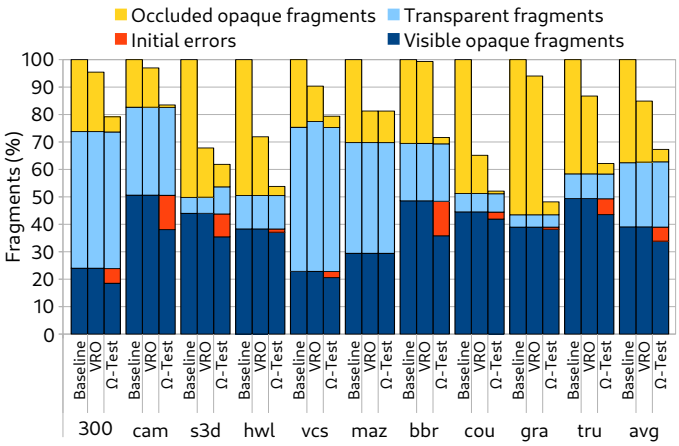


Fig. 12. Breakdown of rendered fragments for the evaluated benchmarks, comparing the the baseline GPU, VRO and Ω-Test.

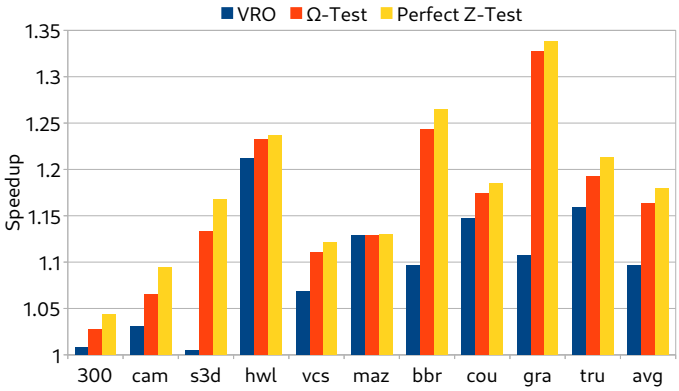


Fig. 13. Speedup comparison (normalized to the baseline GPU) between VRO, Ω-Test and a perfect HSR.

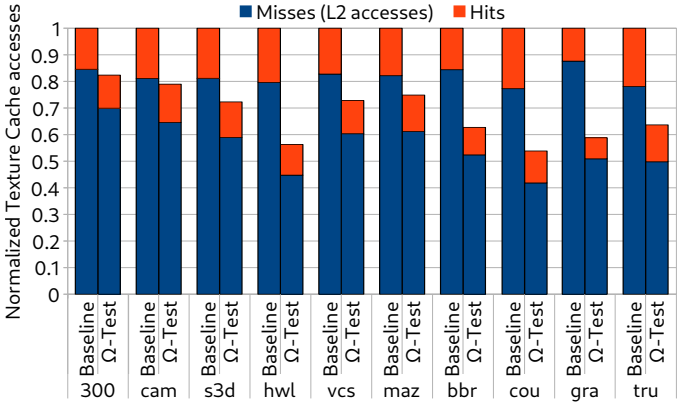


Fig. 14. Texture Cache accesses (normalized to the baseline GPU) broken down into hits and misses that eventually go to L2.

for Gravity –gra–), outperforming VRO which achieves average energy savings of 9.6%. Games such as Vegas Crime Simulator (vcs), Gravity (gra) or Beach Buggy Racing (bbr) achieve high energy savings because of their texture complexity, as explained before, since much of the cost of the overdraw comes from eventual DRAM accesses due to misses in the upper cache levels. Other benchmarks, such as Hot Wheels (hwl) or Counter Strike (cou) also obtain high energy savings (around 18.7%) but not because of the poor caching behaviour of their textures and rather

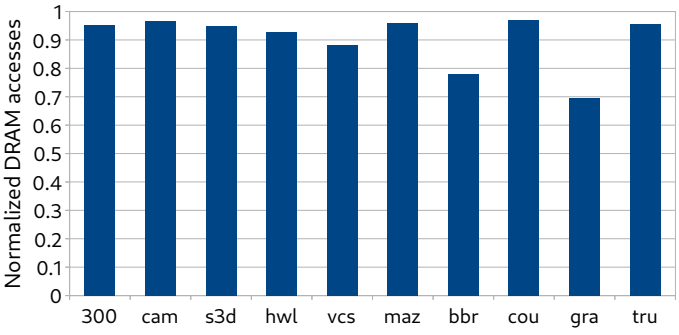


Fig. 15. Total amount of DRAM accesses (normalized to the baseline GPU) for the evaluated benchmarks.

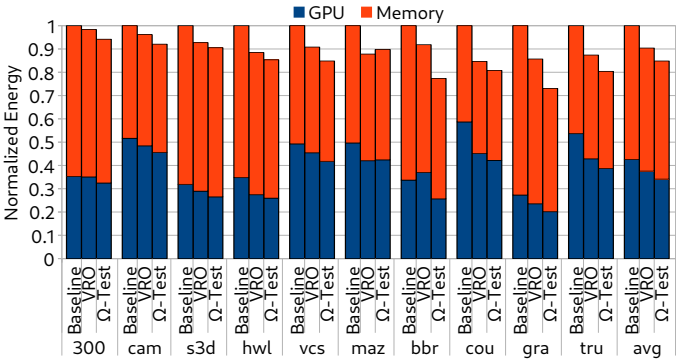


Fig. 16. Energy savings normalized to the baseline GPU.

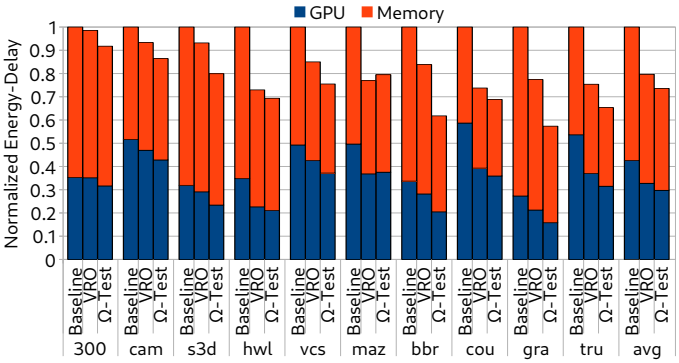


Fig. 17. Energy-Delay product (EDP) savings normalized to the baseline GPU for the evaluated benchmarks.

because of the very high amount of overdraw that is indeed exposed to the baseline TBR architecture, and which our proposal is able to remove.

As per the energy-efficiency of the overall GPU/Memory system, Figure 17 reports the Energy-Delay product (EDP) savings for each benchmark. It can be observed that Ω-Test achieves EDP savings of 26.42% on average (reaching a maximum of 42.65% in the case of Gravity –gra–) whereas VRO achieves EDP savings of 20.34% on average.

7 Conclusions and Future Work

Overdraw plays an important role in the performance and energy efficiency of mobile GPUs, and it is strongly related to the approach used to resolve the visibility of the different primitives. In this work we have proposed the Ω-Test, a novel microarchitecture technique that resolves visibility by using information of the Z-Buffer from the previous frame. We have shown that our approach is much more effective for removing overdraw than a traditional Early Z-Test, which only uses information from the current frame and whose Z-Buffer must be built from scratch every frame.

Our approach relies on frame-to-frame coherence; however, an unexpected depth change in a primitive could potentially lead to an error in the final rendered image. We have included an error detection and correction mechanism to fix the small amount of errors that can appear in certain tiles. Finally, to dramatically reduce the storage needs of the underlying Ω-Buffer, we have also implemented a coarsening mechanism along with the use of an aggregate function, which is highly effective and hardly impacts accuracy. Overall, the Ω-Test reduces the average overdraw of scenes by 32.7%, which results in an average speedup of 16.3% in addition to average EDP savings of 26.42% for a set of commercial representative applications.

As part of the future work, we will consider new compression schemes for the Ω-Table aimed at reducing the memory usage needs as well as improving its accuracy, such as quantization, downsampling and precision reduction. Another path to explore is a per-tile δ margin to more selectively detect objects moving around the scene. These two approaches will improve the efficiency of Ω-Test in terms of performance, energy and area cost. On the other hand, it is common to give control to the applications over features supported by the GPU (Texture samplers, Occlusion queries, depth/stencil tests, Variable Shading Rate, etc.) so they can be tuned to the application needs. With such kind of control, the application could inform through the render-loop, e.g., about changes of the transformation matrices, so Ω-Test could more accurately predict the depth values of the next frame.

Acknowledgments

This work has been supported by the the CoCoUnit ERC Advanced Grant of the EU's Horizon 2020 program (grant No 833057), the Spanish State Research Agency under grant TIN2016-75344-R (AEI/FEDER, EU) and the ICREA Academia program. D. Corbalán-Navarro has been supported by a PhD research fellowship from the University of Murcia.

References

[1] M. C. Shebanow, "An evolution of mobile graphics," https://www.highperformancegraphics.org/wp-content/uploads/2013/Shebanow-Keynote.pdf, accessed June 2021, keynote talk at High Performance Graphics, 2013.

[2] S. Patil, Y. Kim, K. Korgaonkar, I. Awwal, and T. S. Rosing, "Characterization of user's behavior variations for design of replayable mobile workloads," in *International Conference on Mobile Computing, Applications, and Services*. Springer, 2015, pp. 51–70.

[3] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power modeling for gpu architectures using mcpat," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 19, no. 3, p. 26, 2014.

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

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

[6] E. De Lucas, "Reducing redundancy of real time computer graphics in mobile systems," Ph.D. dissertation, Universitat Politècnica de Catalunya, 2018.

[7] J. Bittner and P. Wonka, "Visibility in computer graphics," *Environment and Planning B: Planning and Design*, vol. 30, no. 5, pp. 729–755, 2003.

[8] E. De Lucas, P. Marcuello, J.-M. Parcerisa, and A. González, "Visibility rendering order: Improving energy efficiency on mobile gpus through frame coherence," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 2, pp. 473–485, 2018.

[9] N. Greene, M. Kass, and G. Miller, "Hierarchical z-buffer visibility," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. ACM, 1993, pp. 231–238.

[10] M. Anglada, E. de Lucas, J.-M. Parcerisa, J. L. Aragón, and A. González, "Early visibility resolution for removing ineffectual computations in the graphics pipeline," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 635–646.

[11] T. Akenine-Moller, E. Haines, and N. Hoffman, *Real-time rendering*. AK Peters/CRC Press, 2019.

[12] H. Hubschman *et al.*, "Frame-to-frame coherence and the hidden surface computation: constraints for a convex world," *ACM Trans. on Graphics*, vol. 1, no. 2, pp. 129–162, 1982.

[13] A. Wilson, K. Mayer-Patel, and D. Manocha, "Spatially-encoded far-field representations for interactive walkthroughs," in *Proc. of the 9th ACM international conference on Multimedia*, 2001, pp. 348–357.

[14] 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.

[15] "Photo-realistic deferred lighting," https://www.beyond3d.com/content/articles/19/, accessed March 2021.

[16] N. Thibieroz and W. Engel, "Deferred shading with multiple render targets," *Shader X*, vol. 2, pp. 251–251, 2004.

[17] I. T. Limited, "PowerVR Hardware. architecture overview for developers," http://cdn.imgtec.com/sdk-documentation/PowerVR+Hardware.Architecture+Overview+for+Developers.pdf, accessed August 2019.

[18] N. Greene, "Hierarchical polygon tiling with coverage masks," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996, pp. 65–74.

[19] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff III, "Visibility culling using hierarchical occlusion maps," in *Proc. of the 24th annual conference on Computer graphics and interactive techniques*, 1997.

[20] S. Morein *et al.*, "Ati radeon hyperz technology," http://www.graphicshardware.org/previous/www_2000/presentations/ATIHot3D.pdf, accessed June 2021, Presented at SIGGRAPH/EUROGRAPHICS Workshop On Graphics Hardware, 2000.

[21] M. Andersson, J. Hasselgren, and T. Akenine-Möller, "Masked depth culling for graphics hardware," *ACM Transactions on Graphics (TOG)*, vol. 34, no. 6, pp. 1–9, 2015.

[22] D. Shreiner, "Opengl® programming guide seventh edition," 2010.

[23] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Teapot: a toolset for evaluating performance, power and image quality on mobile graphics systems," in *Proceedings of the 27th International ACM Conference on Supercomputing*. ACM, 2013, pp. 37–46.

[24] "Arm mali-450 gpu," https://developer.arm.com/products/graphics-and-multimedia/mali-gpus/mali-450-gpu, accessed August 2019.

[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," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009, pp. 469–480.

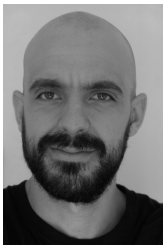
[26] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE computer architecture letters*, vol. 10, no. 1, pp. 16–19, 2011.

[27] "Android sdk," https://developer.android.com/studio, accessed August 2019.

[28] M. Segal and K. Akeley, "The opengl graphics system: A specification (version 1.1)," http://delta.cs.cinvestav.mx/~fraga/Cursos/Graficacion/2007/OpenGL/opengl1.2.1.pdf.gz, accessed June 2021, silicon Graphics, Inc, 1999.

[29] "Gapid," https://developers.google.com/vr/develop/unity/gapid, accessed August 2019.

[30] "Gallium3d," https://www.freedesktop.org/wiki/Software/gallium, accessed August 2019.



David Corbalán-Navarro is a PhD student at the University of Murcia (UMU), Spain. He received his B.S. degree in Computer Engineering in 2016 and his M.S. degree in New Information Technologies in 2017, both from the UMU. He started his PhD in September 2017 when he also joined the ARCO (Architecture and Compilers) research group at the UPC. His main research interest is the architecture of graphics processors with special emphasis on the energy efficiency of mobile GPUs.

Juan L. Aragón is an Associate Professor at the University of Murcia (UMU), Spain. In 2003 he received his PhD degree in Computer Engineering from the UMU, followed by a postdoc at the University of California, Irvine. He has also been a Visiting Researcher at EPFL (Switzerland) and at Princeton University (USA). He has advised 5 PhD theses and co-authored +50 papers in major conferences and journals. His research interests are focused on computer architecture, with special emphasis on heterogeneous systems, application-specific accelerators, microarchitecture, and GPUs.

Martí Anglada received his M.S. degree in High Performance Computing in 2015 and his PhD degree in Computer Architecture in 2020, both from Universitat Politècnica de Catalunya (UPC-BarcelonaTech). He joined the UPC-BarcelonaTech Architectures and Compilers research group in July 2014. His research is focused on energy-efficient architectures for mobile GPUs.

Enrique de Lucas received his B.S. degree in Computer Science in 2010 and M.S. degree in Computer Engineering in 2011 both from Complutense University of Madrid (UCM), Spain. During 2012 he worked on processor microarchitecture at Intel Labs. By February 2013 he joined ARCO research group of Universitat Politècnica de Catalunya (UPC), Barcelona (Spain), where he completed the PhD degree in 2018. His main research interests included techniques to exploit inter-frame coherency and reduce redundancy in the graphics subsystem for increasing the energy-efficiency of GPUs. By January 2017 he joined Esperanto Technologies (Spain). Since March 2020 he is GPU Architect at Imagination Technologies, (UK).

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

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 Architecture and Compilers research group. His research has focused on computer architecture and compilers, with a special emphasis on cognitive computing systems and graphics processors in recent years. He has published over 370 papers, and has served as associate editor of five IEEE and ACM journals, program chair for ISCA, MICRO, HPCA, ICS and ISPASS, and general chair for MICRO and HPCA. He is a Fellow of IEEE and ACM.