A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC

Min Kyu Jeong, Mattan Erez
Dept. of Electrical and Computer Engineering,
The University of Texas at Austin
{mkjeong, mattan.erez}@mail.utexas.edu

Chander Sudanthi, Nigel Paver
ARM Inc.
{Chander.Sudanthi, Nigel.Paver}@arm.com

ABSTRACT

Diverse IP cores are integrated on a modern system-on-chip and share resources. Off-chip memory bandwidth is often the scarcest resource and requires careful allocation. Two of the most important cores, the CPU and the GPU, can both simultaneously demand high bandwidth. We demonstrate that conventional quality-of-service allocation techniques can severely constrict GPU performance by allowing the CPU to occasionally monopolize shared bandwidth. We propose to dynamically adapt the priority of CPU and GPU memory requests based on a novel mechanism that tracks progress of GPU workload. Our evaluation shows that our mechanism significantly improves GPU performance with only minimal impact on the CPU.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*
; I.3.1 [**Computer Graphics**]: Hardware Architecture—*Graphics processors*
; C.3 [**Special-purpose and Application-based Systems**]: *Real-time and embedded systems*
; C.4 [**Performance of Systems**]: *Design studies*

General Terms

Design, Experimentation, Performance

Keywords

Graphics processor, Memory controller, Quality of service, System on chip

1. INTRODUCTION

A modern system-on-chip (SoCs) is typically composed of multiple types of *intellectual property cores* (IP cores) with different functionality. This is done because heterogeneity increases efficiency and decreases development time. All integrated cores share off-chip memory, which is often one of the most constrained resources. High end SoCs, for example, now include powerful CPU and GPU cores, which are both very demanding of the memory system. Optimally allocating the scarce memory bandwidth resource between the CPU and GPU cores is critical yet challenging. The CPU is latency sensitive and cannot tolerate long memory latency without losing performance. The GPU, on the other hand, is designed to tolerate long latencies but requires consistent high bandwidth for periods of time to meet its real-time deadlines. Because the CPU is sensitive to latency, it is common practice to always prioritize requests from the CPU over those of the GPU. We show that such a static policy can lead to an unacceptably low frame rate for the GPU. Conversely, prioritizing GPU requests significantly degrades CPU performance.

We propose a new mechanism to solve this challenge by dynamically adjusting the memory controller's quality-of-service (QoS) policy. As is done today, we prioritize CPU requests by default and GPU requests are serviced opportunistically. When the GPU is expected to miss a deadline, however, we increase the GPU service rate by raising its priority. The key to our technique is identifying when the default policy should be adjusted. We do this by utilizing knowledge of the GPU architecture and monitor the progress of processing a frame against the frame deadline. The memory controller can than determine when a deadline is likely to be missed and boost the GPU service quality.

To the best of our knowledge, we are the first to propose and provide a detailed evaluation of adjusting the memory controller QoS policy in response to progress towards a real-time deadline. We show how the dynamic technique balances both real-time constraints and best-effort memory accesses and maintains GPU target performance with only a small impact on the CPU. We are also first to present a detailed analysis based on a combined cycle-accurate simulation of GPU and CPU cores with a detailed memory system. [1] We draw important insights into how these components interact, which contradicts current best practices.

The rest of this paper is organized as follows: Section 2 provides background on CPU and GPU architecture, memory controllers, and the commonly used QoS mechanisms. Section 3 describes our dynamic QoS mechanisms, based on our technique to monitor GPU's workload progress. We present our evaluation methodology and results in Section 4 and 5, then conclude in Section 6.

2. BACKGROUND AND RELATED WORK

This section briefly discusses the memory access and execution characteristics of CPU and GPU cores, as well as the fundamental principles and design of modern memory controllers and QoS mechanisms.

[1] Concurrent work by Ausavarungnirun et al. [3] also explores the memory-system implications of heterogeneous GPU/CPU processors.

2.1 CPU

Modern general purpose processors are designed mainly to maximize the performance of a single thread of execution. Single-thread performance is very sensitive to long-latency memory requests because instructions dependent on the long latency load cannot proceed until the load completes. Caching and out-of-order execution can mitigate the impact of long main memory latency. Main memory access latency, however, is much higher than what the out-of-order structure can tolerate and cache misses that go out to main memory inevitably stall the accessing thread [7]. Therefore, any increase in CPU memory access latency, such as delays introduced by contention from the GPU, decreases CPU performance.

2.2 GPU

Mobile GPU cores often utilize tile based rendering to reduce off-chip memory bandwidth consumption. The screen is subdivided into many blocks/tiles, which can be processed independently of one another (Figure 1). As the tiles are small enough, the entire pixel data of a tile can be kept in on-chip buffers while being rendered so that repeated accesses to the same pixel do not incur off-chip memory accesses. GPUs can process all vertices and fragments within the tile and multiple tiles in parallel and can therefore tolerate very long memory latencies. They still require high bandwidth and are sensitive to disruptions in available bandwidth.

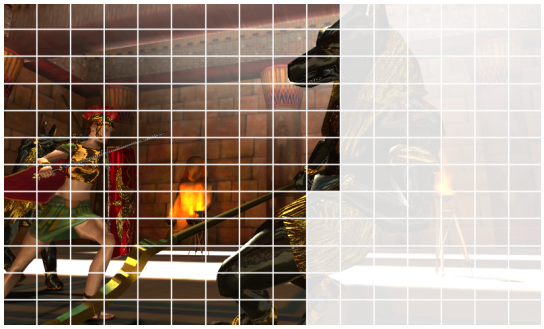


Figure 1: Tile based rendering in progress. Shaded tiles are to be rendered.

Simple scenes can be processed rapidly and generate correspondingly low memory traffic, while others may take the entire time allotted or longer, resulting in skipped frames and degraded user experience. Because frame rate is fixed and frame render time varies, the GPU may idle between finishing a frame and starting the next frame. Figure 2 shows an example of processing one frame from the Taiji GPU workload [1] with two CPU cores running together. The figure shows how the GPU only requires about half the frame time to process a scene (GPU bandwidth consumption shown with dashed lines) and consumes up to 62% of total memory bandwidth when not constrained by the memory controller as discussed below. The figure also shows how the heavy bandwidth use of the GPU hurts CPU performance (CPU's instructions per cycle (IPC) shown with solid lines) compared to when the GPU is idling. The two subfigures show different CPU workloads and the same set of GPU frames. Both `mcf` and `art` are memory-intensive applications from the MinneSPEC suite [9], with `art` requiring somewhat higher bandwidth.

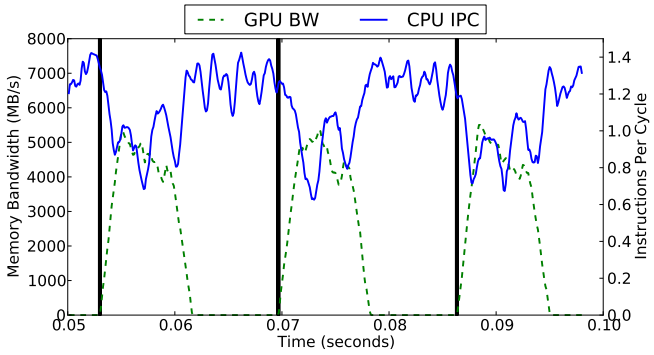
2.3 Memory Controller Basics

Modern DRAM architecture is optimized for access patterns with spatial locality. Within a DRAM chip, each access is performed at a granularity of an entire row, which is 8Kb or 16Kb in current technology [10]. To amortize the time and energy involved in activating an entire DRAM row, each DRAM bank contains a *row buffer*. Consecutive accesses to the same row (row buffer hits) can be served directly from the row buffer, saving time and energy in activating rows. In contrast, accesses to a different row (row buffer misses) need additional steps of precharging the array and activating the new rows. When the row buffer hit rate is low, DRAM can only supply a small fraction of its peak bandwidth. Therefore, modern out-of-order memory controllers schedule accesses to the same row together by prioritizing row buffer hit requests [13].

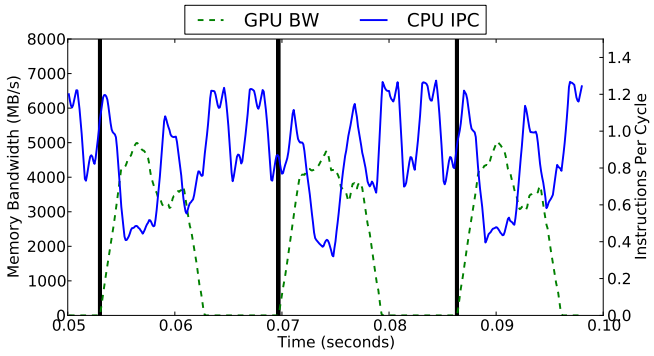
2.4 Quality of Service

An out-of-order memory scheduler increases overall bandwidth, but in a shared memory SoC, the priority scheme can starve some cores when other cores offer frequent requests with high spatial locality, like GPUs do. To prevent such unfairness, the memory controller must balance the accesses from different cores and provide QoS mechanisms. Because of the heavy competition in the SoC industry, very little information on how commercial SoCs manage shared memory bandwidth is publicly available.

Most previous literature on off-chip memory bandwidth QoS focuses on a different context than our multi-processor SoC (MPSoC), such as real-time systems and general purpose chip multiprocessors (CMP). High-end SoCs combine both real-time and best-effort components and place very



(a) CPU cores: `mcf-art`, GPU unconstrained



(b) CPU cores: `art-art`, GPU unconstrained

Figure 2: GPU activity (bandwidth consumption) and CPU performance. Vertical lines represent frame deadlines. Experimental setup discussed in Section 4.

high pressure on shared memory bandwidth. Prior work on QoS for CMPs (e.g., [11, 12]) does not consider real-time constraints, thus can lead to an unacceptable rate of missed deadlines for the GPU. Work on real-time systems, on the other hand, has focused exclusively on bounding the latency of individual requests and ensuring a minimal fraction of shared throughput [2]. This approach sacrifices effective memory scheduling in favor of guaranteed deadlines, which leads to very poor utilization of available DRAM bandwidth and requires significant over-provisioning of this scarce and expensive resource. Recent white papers [15, 16] discuss general quality-of-service techniques and recommendations and appear to describe the status quo. This status quo is that two techniques are effective when combined: regulating the number of outstanding GPU requests and prioritizing CPU requests over ones from the GPU. Note that previous academic literature does not address this particular problem of sharing bandwidth between best-effort and real-time workloads from different cores.

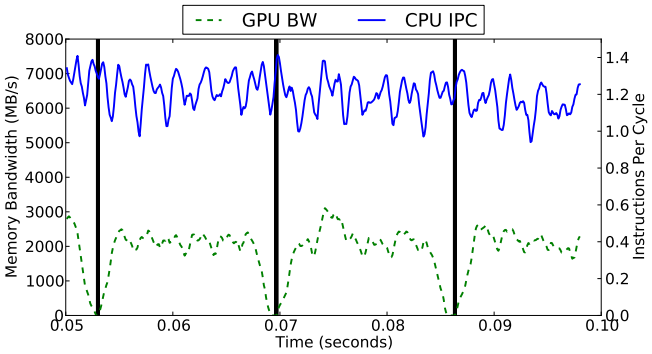
Restricting the number of outstanding GPU requests reduces the GPU's ability to continuously send requests to the memory system, even though the abundant parallelism associated with graphics allows many concurrent requests. The smaller the number of outstanding GPU requests, the greater the number of memory access issue slots that are available for other cores. There are several equivalent mechanisms that can be used to constrain the number outstanding GPU requests, including separate memory controller queues for each core, which may be either physical or virtual.

Guaranteeing available request queue slots is insufficient because the memory controller may still prefer to always issue GPU requests. To avoid this situation, an age-based QoS technique can be used [11]. Recent guidelines, however, suggest that CPU requests should receive higher priority to decrease the performance lost to contention-induced high latency memory accesses [15, 16]. While the priority policy is static, the GPU may take advantage of much of the available bandwidth when CPU cores do not access main memory frequently.

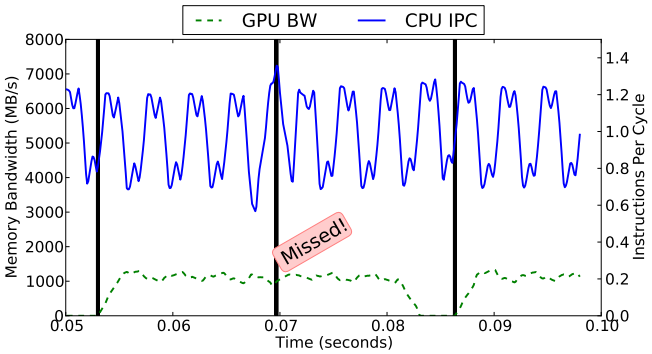
Prioritizing CPU requests indiscriminately, however, can hurt GPU performance significantly, when a CPU core continuously uses high memory bandwidth and GPU workload is complex enough to mandate high memory bandwidth as well. The impact of the aggressive QoS is shown in Figure 3. The figure shows the same workload scenario as in Figure 2, but with a QoS mechanism that balances memory performance by restricting the GPU to 8 outstanding requests and prioritizing all latency-sensitive CPU requests. When compared to Figure 2, the QoS mechanisms successfully prevent CPU starvation and CPU performance is not impacted by the GPU. With this static QoS, however, the GPU suffers. When `mcf` and `art` are run together with the GPU, the GPU receives barely enough bandwidth to maintain the frame rate. With the higher bandwidth `art-art` CPU workload, frame deadlines are missed.

From this example and discussion, we can conclude that for the static QoS scheme to generally work for any workload scenario, the memory bandwidth needs to be overprovisioned for the worst case. Otherwise, it is possible that frames must be dropped, either while reconfiguring or while programming. A better alternative to costly overprovisioning, is to identify when the GPU should be allowed to nearly monopolize bandwidth to meet its real-time constraints, which we discuss in the next section.

3. DYNAMIC QUALITY-OF-SERVICE



(a) CPU cores: `mcf-art`, GPU out=8, cpu > gpu



(b) CPU cores: `art-art`, GPU out=8, cpu > gpu

Figure 3: GPU activity (bandwidth consumption) and CPU performance. GPU is restricted to at most 8 outstanding memory requests and CPU requests are given higher priority. Vertical lines represent frame deadlines. Experimental setup discussed in Section 4.

Static QoS mechanisms lack the ability to adapt to the dynamic behavior of real workloads, resulting in either degraded CPU performance (Figure 2), or missed GPU deadlines (Figure 3(b)). In order to achieve high CPU performance while satisfying real-time constraints, we propose to dynamically adjust the QoS policy based on runtime workload characteristics. Ideally, CPU requests should be prioritized as long as the CPU does not compromise the GPU target frame rate. The key to achieving behavior that is near this ideal is to identify when a deadline is likely to be missed and only then adjust the QoS policy and either treat the GPU and CPU as equals, or even prioritize GPU requests. We discuss how to predict when the GPU makes insufficient progress and a heuristic to adjust priority below.

3.1 Monitoring GPU workload progress

As discussed in Section 2.2, mobile GPUs typically partition the screen into equal-sized tiles and process them in order. Each tile is processed once for all primitives that overlap it, then it is not accessed again until the following frame. We exploit this to track the progress the GPU is making in the current frame. The GPU hardware is aware of how many tiles in total it must process, the order in which tiles are processed, and what tiles are currently active. Progress is thus simply the current position within the total frame, as described by Equation 1. This information can readily be communicated from the GPU to the memory controller to affect the QoS policy.

$$FrameProgress = \frac{Number\ of\ tiles\ rendered}{Number\ of\ tiles} \quad (1)$$

Although this progress monitoring mechanism is simple, it is very effective in our system because the mobile GPU uses fine-grained tiles. With coarser tiles or non-tiled GPU architecutres, a more sophisticated estimation of workload can be used, such as those suggested by prior work in the context of coarse-grained adjustments to the GPU voltage and frequency [5, 14] [2] .

3.2 Dynamic QoS policy

To determine the QoS policy, the memory controller compares the frame progress rate, obtained above, with the *expected progress rate*. The expected progress rate can be calculated by dividing the elapsed from the beginning of the frame by the target frame time, (e.g. $16.67ms$ for 60 frames-per-second (FPS)) as shown in Equation 2. As with tracking progress, more sophisticated techniques can be used to obtain higher accuracy estimates of expected progress [5].

$$ExpectedProgress = \frac{Time\ elapsed\ in\ current\ frame}{Target\ frame\ time} \quad (2)$$

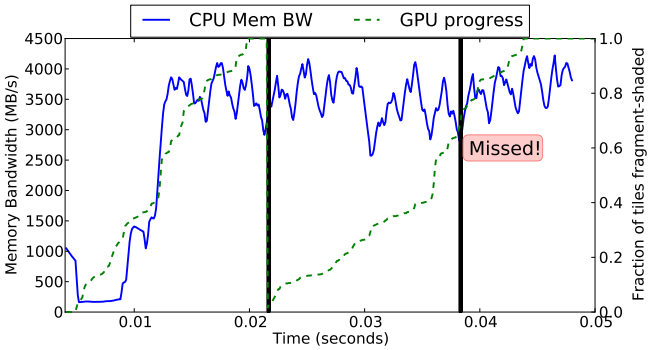
The memory controller then chooses a QoS policy based on how far the GPU is behind its expected progress point. Algorithm 1 shows an example dynamic QoS policy, which we employ in this paper and which works well in our experiments. There are two priority levels, and the CPU gets the higher priority as long as the current GPU progress rate is above the expected rate. When the progress falls behind the expected rate, GPU priority is increased to equal that of the CPU. When only 10% of the frame time remains until the deadline and if the GPU has not yet caught up to its expected point, the GPU is prioritized above the CPU in an attempt to make the frame deadline. This 10% buffer was chosen arbitrarily and can be tuned for better performance. Again, we favored a simple design that can demonstrate the benefits and importance of the dynamic approach. We leave refinements of this QoS selection algorithm to future work.

Algorithm 1 Dynamic QoS policy

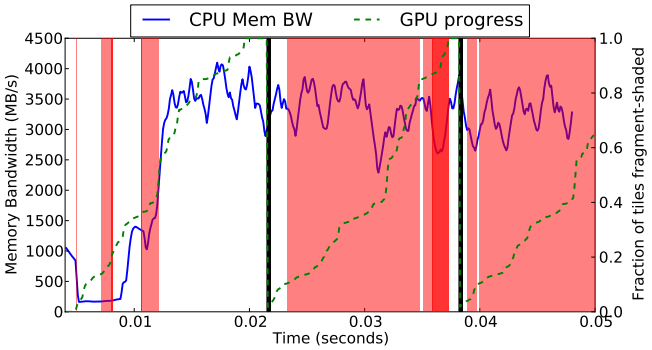
```
if FrameProgress > ExpectedProgress then
    CPU_priority = High
    GPU_priority = Low
else if ExpectedProgress > 0.9 then
    CPU_priority = Low
    GPU_priority = High
else
    CPU_priority = GPU_priority
end if
```

Figure 4(a) demonstrates how a static mechanism that prioritizes the CPU can lead to a missed GPU deadline. With our dynamic scheme, GPU priority is dynamically increased to enable it to meet its deadlines (Figure 4(b)). In the first frame, the GPU makes acceptable progress most of the time even with CPU priority. In the second frame, however, the GPU requires equal priority for much of the frame and higher priority towards the end of the frame to ensure the deadline is met.

[2]Prior work estimates workload at frame granularity, and does not discuss monitoring in-frame progress dynamically. An additional hardware counter is needed to keep track in-frame progress, such as number of geometries processed. Our tile-based monitor does not need any additional hardware, as the tile bookkeeping is an integral part of the GPU's job management.



(a) Static QoS leading to missed deadline



(b) Dynamic QoS adjusting priority to make deadline

Figure 4: CPU memory bandwidth consumption and GPU progress over several frames for static and dynamic QoS. Time intervals shaded in light red indicate times that GPU progress was insufficient and CPU and GPU priority are equal. Dark red shading indicates the critical periods of time that the dynamic scheme prioritizes GPU requests over CPU accesses.

4. EVALUATION METHODOLOGY

We evaluate our proposed dynamic QoS scheme using cycle level simulations. We use a combination of the gem5 system simulator [4], a proprietary next-generation GPU simulator, and the DrSim DRAM simulator [6]. The gem5 out-of-order CPU model and the GPU model share the DRAM model through the gem5 bus. DrSim models memory controllers and DRAM modules faithfully, simulating the buffering of requests, scheduling of DRAM commands, contention on shared resources (such as address/command and data buses), and all latency and timing constraints of LPDDR2 DRAM.

System configuration

The QoS schemes we simulate include uncontrolled CPU and GPU (`noqos`), static CPU priority over GPU (`static`), and our dynamic scheme (`dynamic`). Constraining the number of outstanding GPU requests to N (`outN`) is used in combination with `static` and `dynamic`.

Table 1 summarizes the system parameters of our simulated systems. We believe the simulated system is representative of the next-generation high-end mobile SoC. Memory scheduling queues are large enough to guarantee room for

CPU	Dual-core, 1.2GHz ARM out-of-order superscalar
Caches	32KB private L1 I/D, 1MB shared L2
GPU	8 Unified shader cores, 600MHz
GPU L2	128KB shared
System bus	128-bit wide, 1GHz
Memory controller	FR-FCFS scheduling, open row policy 64 entries read queue, 64 entries write queue
Main memory	1 channel, 1 rank / channel, 8 banks / rank 4 x16 LPDDR2-1066 chips / rank 8.3GB/s peak BW, All chip parameters from the latest Micron datasheet [10] XOR-interleaved bank index [17]

Table 1: Simulated system parameters

GPU workload	Source	Target FPS
`taiji`	3DMarkMobile ES 2.0 [1]	60
`egypt`	GLBenchmark [8]	60
`taiji1080p`	3DMarkMobile ES 2.0	30
`farcry`	Game	30
CPU workload	**Average Mem Bandwidth**	
`art-art`		5.8GB/s
`mcf-art`		3.5GB/s
`mcf-mcf`		800MB/s

Table 2: Workloads used and their characteristics.

CPU requests even when GPU was not constrained in `noqos`.

Workloads

Due to the slow GPU simulation speed, it is impractical to run a GPU accelerated application and other memory intensive applications on top of the full OS and GPU driver stack. Instead, we run CPU workloads on the CPU cores in parallel with graphics workloads on the GPU to approximate the memory bandwidth constrained usage scenario.

Table 2 shows the CPU and the GPU benchmarks used. We selected two SPEC CPU 2000 benchmarks with the MinneSPEC input set [9], which place significant demand on the memory system. Dual-core CPU workloads are multi-programmed to simulate 3 levels of CPU memory bandwidth usage. GPU workloads are post-driver output of a representative frame from each graphics benchmark; `taiji` and `egypt` are WVGA resolution and `taiji1080p` and `farcry` are 1080p. Their target performance in frames-per-sercond (FPS) was determined by measuring the their execution time on GPU without CPU interference. Two workloads, `taiji1080p` and `farcry` were not able to finish in 16.67ms on our simulated system, but finished within 33.34ms. We assume that missed frames are skipped and the behavior is repetitive, so the target FPS is an integer divisor of the 60 FPS base.

5. RESULTS

In this section we present experimental results that demonstrate the effectiveness of the dynamic mechanism. We focus on challenging workloads that are constrained by the available memory bandwidth of the SoC. In such cases, it is impossible to simultaneously meet the target frame rate and service the CPU without GPU interference. We analyze the interaction between the components and show how dynamic can adapt to the changes in workload demand combination and provide the near-optimal QoS strategy, while current guidelines for static QoS fail.

To better quantify these interactions, we show the results of multiple QoS schemes for GPU and CPU performance in Figure 5 and Figure 6, respectively. The results point to two important insights, which contradict the current best practice of prioritizing the CPU and constraining the GPU.

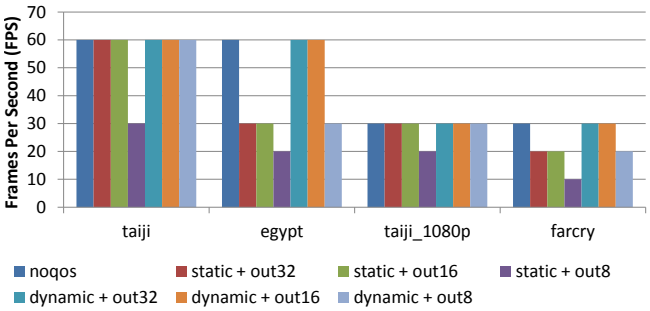


Figure 5: GPU performance in frames per second (FPS) when both CPU cores run `art`.

First, restricting the number of outstanding GPU requests can cause the GPU to miss deadlines, while at the same time often *degrading* CPU performance. As shown in Figure 5, restricting the GPU to 8 outstanding requests reduces the frame rate by 33% or 50%, dropping one of every three or two frames respectively. The impact on the CPU is interesting. Figure 6 shows that as long as the GPU meets its deadlines (configurations for which the GPU fails are shaded black in Figure 6) the CPU either sees little benefit from constraining the GPU, or experiences noticeable performance degradation. We found that having fewer GPU requests for the memory scheduler to choose from prevents efficient scheduling and reduces the effective memory bandwidth. Both the GPU and CPU cores suffer from the longer low effective bandwidth period.

The CPU benefits from a constrained GPU only when the GPU fails to meet its required frame rate (black bars). For example, restricting the GPU to 8 outstanding accesses leads to only a 5% degradation in CPU performance in `farcry-art-art`. The GPU however, only achieves 10 FPS instead of the targeted 30 FPS. This is generally unacceptable.

Second, our GPU progress-aware `dynamic` QoS mechanism can indeed adapt to the changes in CPU and GPU workloads and provide the performance of the best QoS setting for each workload. When bandwidth is not sufficient for the workloads (`egypt` and `farcry` in Figure 5), only `noqos` and `dynamic` meet GPU performance requirements. Even in such severely bandwidth-constrained cases, `dynamic` can still find opportinity to give the CPU some priority and reduces slowndown by 3.6% and 6.9% from `noqos`. When there is sufficient bandwidth to serve the GPU and CPU cores simultaneously, CPU performance of `dynamic` +`out32` roughly matches the best `static` configuration of each workload (shown as the lowest non-black bar in Figure 6) and provides up to 9.4% reduction in slowdown from `noqos`.

Again, constraining the outstanding number of requests from the GPU hurts CPU performance even more for `dynamic` with `egypt` and `mcf-art`. It turns out that the constraint slows down GPU progress and our `dynamic` scheme forces the memory controller to raise GPU priority, and therefore the CPU suffers. Since `mcf-art` uses a moderate amount of memory bandwidth, GPU requests can be issued opportunistically even with low priority. Therefore, having many outstanding requests at the memory controller ready allows them to be scheduled efficiently and enables the GPU to make good progress. In the `dynamic` + `out32` configuration, the memory controller doesn't raise GPU priority and the CPU gets low-latency priority accesses, yielding better performance.

6. CONCLUSION

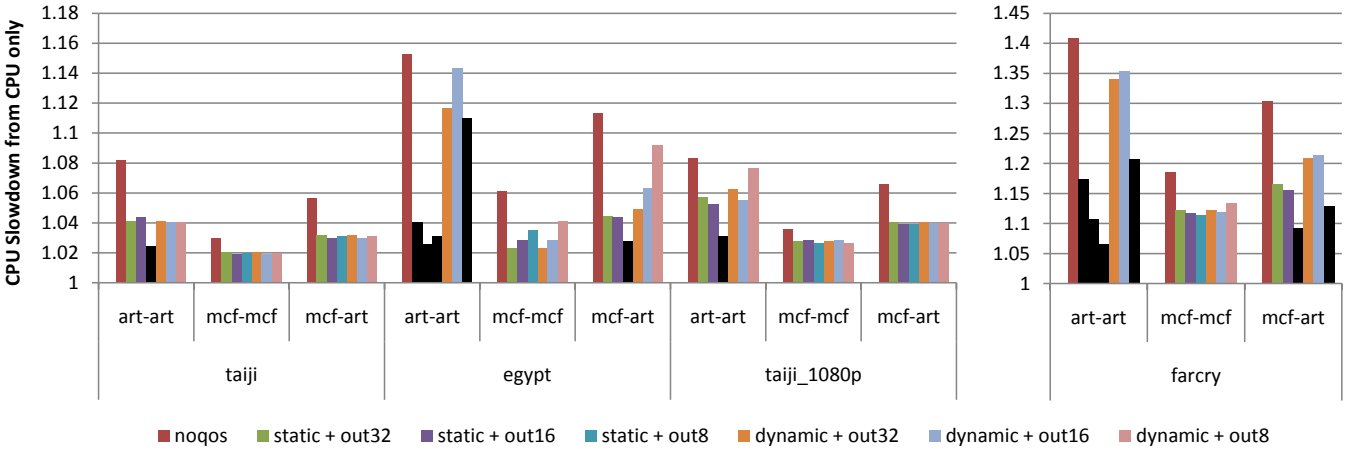


Figure 6: Slowdown of the CPU relative to its performance in an SoC with the same memory configuration but no GPU. Black bars represent configurations in which the GPU could not meet the workload's performance target.

In this paper we carefully analyzed the performance of a system that is representative of current and upcoming advanced SoCs. We used a cycle-level simulator that accurately models an SoC with two CPU cores and a mobile GPU, which all share a single DRAM main memory system. By evaluating the complex interactions between these components we show that current best-practice QoS mechanisms are insufficient and often apply the wrong QoS policy. We determine that there is no single static policy that can be used to simultaneously meet the requirements of the GPU without significantly impacting the CPU cores.

We use this insight to develop a dynamic QoS scheme that maintains CPU priority when possible, but shifts priority towards the GPU if it predicts that the GPU will miss a real-time deadline. We propose a simple, yet effective, tile based frame progress tracking mechanism to enable dynamic QoS policy decisions and show that it both enables the GPU to meet its deadlines and minimizes impact on the CPU. Using our technique, we also conclude that restricting the number of outstanding GPU requests, a static QoS mechanism in use today, often degrades the performance of all cores, because it limits the ability of the memory scheduler to exploit locality.

These conclusions are important and open the way to additional research. This paper used a GPU and CPU as an example of cores with conflicting demands: latency-sensitive best-effort CPU cores, and a bandwidth-sensitive real-time GPU. These diverse requirements are common and a growing number of cores share an increasingly constrained memory system. We believe dynamic techniques, such as the one we present, are the key to enabling such future systems to meet user requirements while still efficiently utilizing scarce shared resources.

7. REFERENCES

[1] 3DMarkMobile ES 2.0. `http://www.futuremark.com/products/3dmarkmobile`, 2011.

[2] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A Predicatable SDRAM Memory Controller. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis - CODES+ISSS '07*, page 251, New York, New York, USA, Sept. 2007. ACM Press.

[3] R. Ausavarungnirun, G. Loh, K. Chang, L. Subramanian, and O. Mutlu. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *Proc. the 39th Ann. Int'l Symp. Computer Architecture (ISCA)*, ISCA '12, New York, NY, USA, 2012. ACM.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39:1–7, Aug. 2011.

[5] Y. Gu and S. Chakraborty. A Hybrid DVS Scheme for Interactive 3D Games. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–12. IEEE, Apr. 2008.

[6] M. K. Jeong, D. H. Yoon, and M. Erez. DrSim: A platform for flexible DRAM system research. `http://lph.ece.utexas.edu/public/DrSim`.

[7] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *Workshop on Memory Performance Issues*, 2002.

[8] Kishonti Informatics Ltd. GLBenchmark. `http://www.glbenchmark.com`, 2011.

[9] A. J. KleinOsowski and D. J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *IEEE Comput. Archit. Lett.*, 1:7–, January 2002.

[10] Micron Corp. *Micron 2 Gb ×16, ×32, Mobile LPDDR2 SDRAM S4*, 2011.

[11] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *International Symposium on Microarchitecture*, pages 146–160, 2007.

[12] K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith. Fair queuing memory systems. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 208–222. IEEE Computer Society, 2006.

[13] S. Rixner, W. J. Dally, U. J. Kapasi, P. R. Mattson, and J. D. Owens. Memory access scheduling. In *Proc. the 27th Ann. Int'l Symp. Computer Architecture (ISCA)*, Jun. 2000.

[14] B. Silpa, G. Krishnaiah, and P. R. Panda. Rank based dynamic voltage and frequency scaling for tiled graphics processors. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 3–12, New York, NY, USA, 2010. ACM.

[15] A. Stevens. Qos for high-performance and power-efficient hd multimedia. Technical report, Arm, 2010.

[16] A. Tune and A. Bruce. How to tune your SoC to avoid traffic congestion. In *DesignCon*, 2010.

[17] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proc. the 33rd IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, Dec. 2000.