WHITE PAPER

Optimizing Performance for an ARM Mobile Memory Subsystem

Ashwin Matta
Senior Product Marketing Manager, Systems and Software Group, ARM

Introduction

Contemporary mobile platform SoCs impose intense traffic management demands on the memory subsystem. An intelligent memory controller design comprehends the fundamental memory streaming requirements of a mobile SoC and provides the necessary capabilities for optimal Quality of Service (QoS) while ensuring best use of available memory bandwidth. This paper describes some of the performance challenges for memory subsystems in an ARM®-based mobile SoC*. Memory controller features necessary for optimizing performance of mobile traffic are described along with their effects, using benchmarking data. Moreover, the combined effect of optimizing memory subsystem performance by closely integrating both the memory controller and the interconnect fabric is demonstrated.

*(*In the context of this paper, the word 'mobile' somewhat loosely refers to all types of systems that deploy LPDDR4/3 or DDR4/3n memories, ranging from smartphone, tablets, laptops/clamshells to systems in consumer and automotive devices.)*

ARM Mobile Subsystem Example

Figure 1 shows a contemporary example of an ARM-based mobile subsystem. Typically, there are one or two clusters of Cortex®-A processors in big.LITTLE™ configuration – with the 'big' CPUs handling the raw computational needs whereas the 'LITTLE' ones running the lighter threads for power efficiency. The CPUs seamlessly communicate data with each other over a Cache Coherent Interconnect, CoreLink™ CCI-550, that provides a snoop filter for storing a directory of cached data, thereby reducing number of snoops required across CPU clusters. In addition to the CPUs, graphics, video and display computations are performed by the fully-coherent Mali™ Mimir GPU and non-coherent V550 Video and DP650 Display processors in the system.

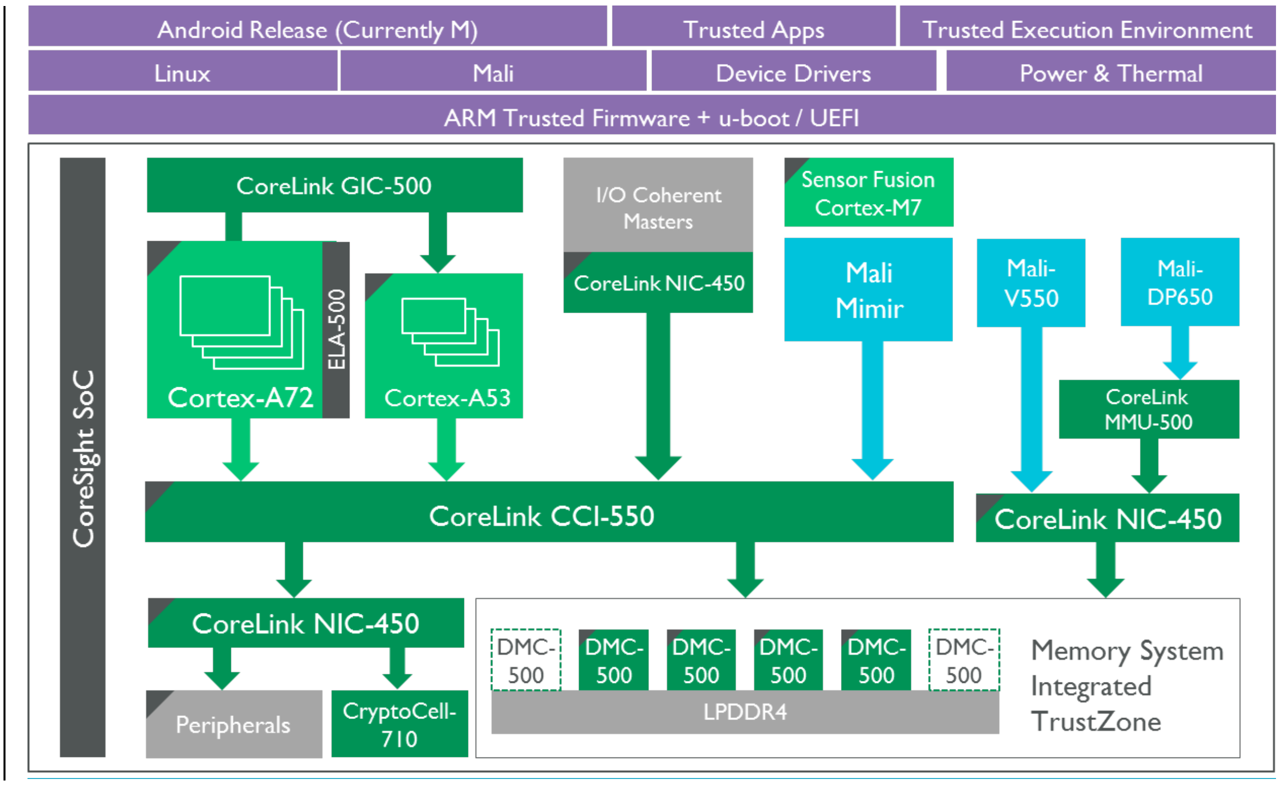


Figure 1: Example ARM-based mobile system

As a side note, having a fully-coherent GPU enables Shared Virtual Memory between heterogeneous processing units – the CPU and the GPU – allowing memory pointers to be passed between these units for true heterogeneous processing. This in itself improves overall performance by enabling these powerful compute blocks to operate on the same block of memory down to the byte level. For further information on this topic, refer to whitepaper from Tirias Research: http://www.tiriasresearch.com/downloads/arm-enables-heterogeneous-computing-the-corelink-cci-550-and-dmc-500/

GPUs and other coherent IO agents such as co-processors, camera streams, etc. share data with CPUs over the CoreLink CCI-550. In order to have the same view of physical memory as the CPUs, accesses by these agents are translated by a Memory Management Unit (MMU), which also optionally provides a Stage2 translation for supporting virtualization. The Video and Display agents communicate data over a non-coherent hybrid interconnect, CoreLink NIC-450, but are also subject to the same MMU translations seen by the IO coherent agents. MMU translation tables are stored in memory and cached locally to minimize the overhead of fetching the translations from memory for each agent communicating with memory.

A complex system such as the one showed in Figure 1 would typically have a memory subsystem consisting of four to six channels of memory controllers, CoreLink DMC-500, connected to LPDDR4/3 memories via an external DFI-compatible PHY. At the maximum supported data transfer rate of LPDDR4-4266 Mbps, the maximum bandwidth to memory would thus be 51.2 GB/s using 6 channels of x16 memory. This memory data bandwidth is capable of supporting data-hungry devices such as advanced tablets and clamshells while continuing to support the extremely low power budgets that ARM is known for.

Traffic Optimization by Mobile Memory Subsystems

Although the presence of a CCI in mobile SoCs reduces the number of accesses required to memory, the sheer number of active agents in the overall system results in high-bandwidth demand for memory data. For the purpose of this analysis, we make a common-scenario assumption that the agents can be collectively classified into 3 categories:

1. Low Latency (LL) – the dominant characteristics of memory traffic coming from the CPUs are random, small size accesses (typically cache line fills) that are sporadic in nature. Key requirement for CPU accesses is low latency so as to provide maximum thread execution performance.
2. Real Time (RT) – typically these agents are Video and Display units which need to access data from memory at a guaranteed rate so as to ensure the real time performance of the mobile device. Any interruptions in data access from memory could result in noticeable flickering in the display thereby rendering the device defective.
3. High Bandwidth (HB) – the GPU and external co-processors are the best example of agents that require high bandwidth access to memory. In the absence of LL or RT traffic, HB traffic can easily consume the entire available bandwidth to memory.

Note that it is not necessary for CPU traffic to always be low latency. Multi-modal latency requirements for CPU traffic are plausible but don't necessarily change the discussions presented in this paper.

By classifying key memory activity in a mobile SoC into the LL, RT and HB categories, the primary mobile memory subsystem optimization problem can be summed up as follows:

"The memory controller needs to provide the lowest latency for LL agents while ensuring that RT agents get guaranteed rate of access to memory to meet real-time performance requirements – and do all this while ensuring that all remaining bandwidth after servicing LL and RT agents is available to the HB agents for 100% total memory bandwidth utilization."

ARM's Memory Performance Optimization Techniques

ARM has been delivering memory controller IP to its customers for several years now. Each new memory generation has resulted in a new controller micro-architecture to match increasing memory transfer speeds and supporting advanced features. Over the years, the various micro-architectural techniques have built on top of the base architecture to provide the best performance achievable with the ARM IP ecosystem.

Prior to a deep dive into various techniques for achieving fine grain control over memory access requirements for different agents, it is necessary to benchmark the bandwidth utilization of the memory controller across a varying number of banks and sizes of accessed data. This is to ensure that the scheduler is robust enough for any system traffic scenario.

Figure 2 shows a plot of the same. The red grid is the theoretical maximum utilization achievable for any memory controller assuming ideal operation with memory access timing restrictions imposed by standard JEDEC timing parameters. The green heat map is the measured bandwidth utilization of CoreLink DMC-500. The measured utilization tracks the theoretical maximum very closely (>90% match). There are no local 'hot spots' indicating that the scheduler for this memory controller is flexible enough for handling the wide range of operating conditions with minimal bandwidth loss.

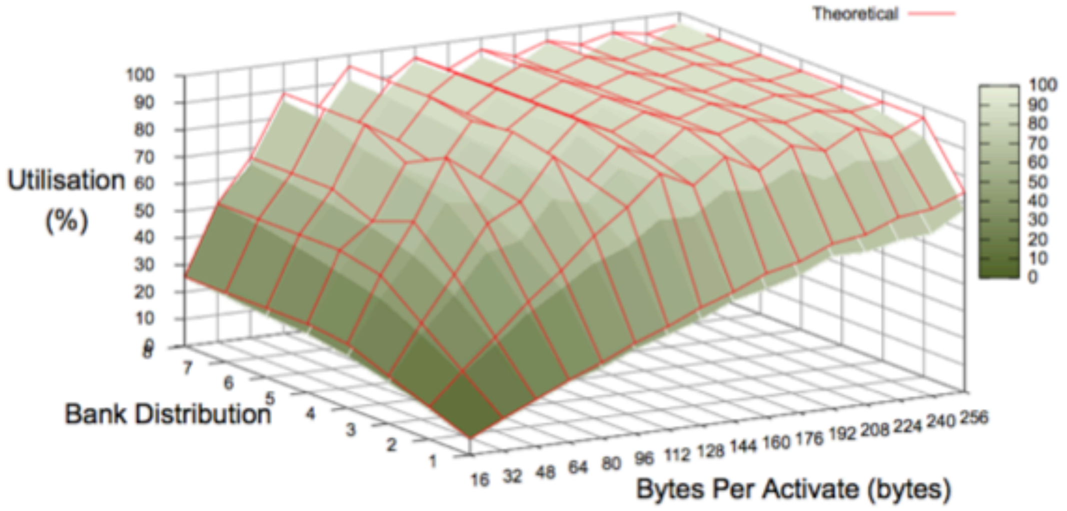


Figure 2: Measured Bandwidth Utilization VS Theoretical Maximum

Building up further on this analysis is a description below of the various memory performance optimization techniques:

1. QoS Arbitration – One of the simplest methods to provide Quality of Service for various agents trying to access memory is through priority arbitration whereby simultaneous incoming memory transactions get serviced based on assigned priorities for the respective agents. Figure 3 shows the effects of QoS arbitration for 2 types of systems. In non-congested systems, latency response to transactions at various priority levels is nearly identical through the range of transactions. Conversely, in congested systems, latency response is stretched out (longer latencies) for transactions from RT and HB agents whereas those from LL agents get a faster response.

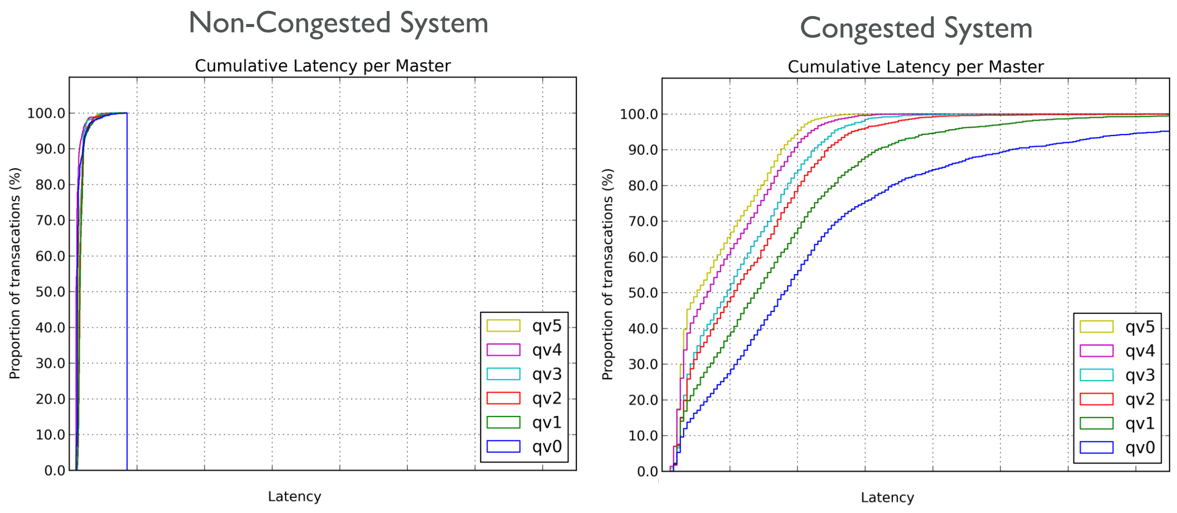


Figure 3: Effect of QoS Arbitration on Congested and Non-Congested System

2. Priority Escalation – A system that remains congested for long periods of time runs the risk of starving execution of low priority transactions in favor of high priority ones. A well-designed QoS scheme must therefore provide a mechanism for priority escalation based on aging counters or programmable overrides. Figure 4 shows the effect of priority escalation in a congested system. Latency response for

QoS level qv0, qv1, etc. which are lower priority transactions improves with escalation. Thus a system can be designed to provide an appropriate latency response across all transfers at various QoS levels.

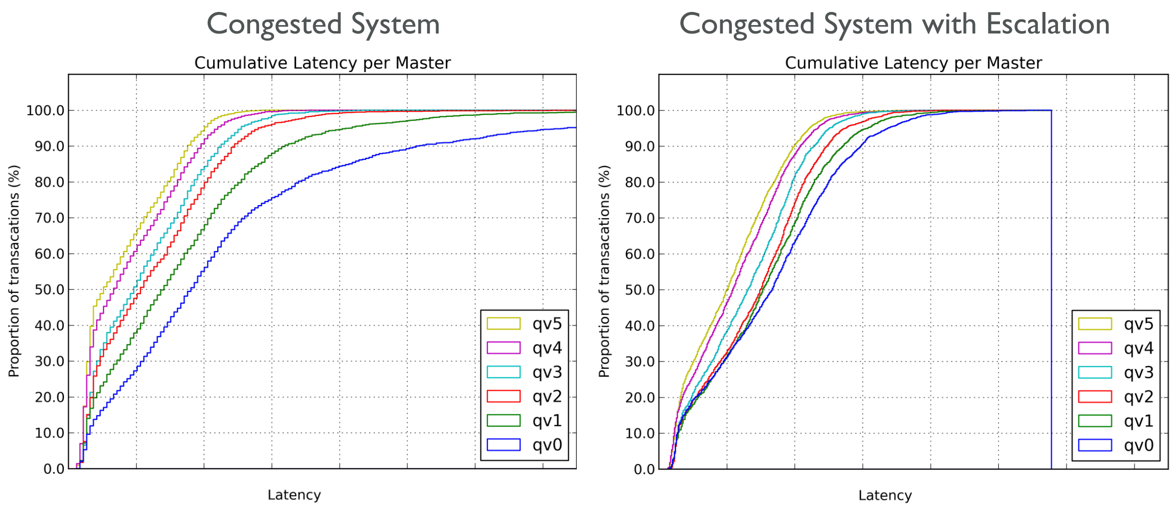


Figure 4: Effect of Priority Escalation in a Congested System

3. Latency Deadline Scheduling – Whereas both QoS arbitration and priority escalation are good levers for managing latency responses for LL, RT and HB agents, they lack the fundamental requirement of timeout or deadline scheduling. This requirement is particularly necessary for RT agents where guaranteed response within a maximum number of cycles is necessary for proper operation of the mobile device, regardless of transaction activity occurring in the memory controller. Figure 5 shows the effect of deadline scheduling on latency response of RT agents. In this scenario, a latency deadline of 'D' cycles has been programmed for the RT traffic. For latencies well below the deadline threshold, the LL transactions get higher priority and have a steeper latency response for nearly 95% of the LL transactions. Closer to the latency deadline, the latency response of RT agents picks up with 100% of the RT transactions completing before the deadline – ahead of the LL transactions.

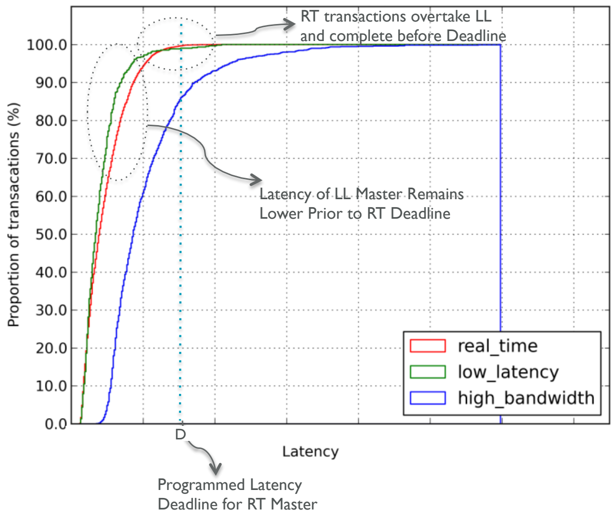


Figure 5: Effect of Latency Deadline Scheduling

4. Deadline Arbitration with Increasing Traffic – The techniques described above are now applied to an illustrative scenario to test the extreme limits. The traffic from LL and RT agents is continuously increased from 0 to 100% injection rates, while the HB agents are also sending transactions through to the memory controller. Figure 6 shows the combined effect of all these controlling mechanisms operating simultaneously. At very low injection rates, LL transactions get their guaranteed bandwidth (~10% of total bandwidth) due to their higher priority. HB traffic utilizes the bulk of the remaining bandwidth as RT traffic is still small. With increasing injection rates, LL transactions continue to get as much bandwidth as requested but HB transactions start stalling in favor of RT transactions which need a guaranteed response time due to deadline scheduling. This trend continues with increasing rate of injection of RT traffic. At around 90% injection rate, the LL transactions can no longer continue getting their guaranteed bandwidth despite having the highest priority because the rate of RT injection is so high that deadline scheduling arbitrates over priority. Bandwidth utilization by HB transactions drops to 0% as they have no mechanism to override LL and RT transactions. At the tail end, near 100% injection rates, essentially all the bandwidth is allocated to RT transactions. A minuscule amount is left for the LL transactions on account of the memory controller scheduler finding opportunity holes between RT transfers to sneak in an LL transaction. The total delivered system bandwidth is consistently high throughout. Thus agent QoS demands are being met without compromising on overall system bandwidth.

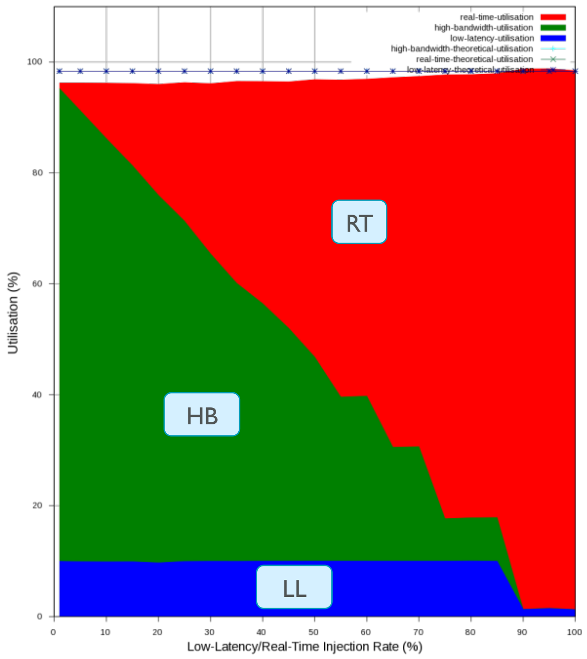


Figure 6: Deadline Arbitration with increasing LL and RT Traffic Rates

Under normal operating conditions, LL/RT injection rates in a mobile system do not exceed 60%. Under these conditions, LL traffic is guaranteed its ~10% bandwidth whereas the rest is distributed between RT and HB traffic.

5. Bus Turnaround Time – An important control knob associated with the memory controller scheduler is the number of cycles after which the DRAM bus is released upon completing a series of transactions from a single agent. Once released, other agents get access to the bus and can start piping through their transactions. Context switching of the bus from one agent to another is not necessarily efficient for bandwidth utilization. Figure 7 shows the two ends of the spectrum from fast bus release (low bus turnaround time) to slow bus release (high bus turnaround time). Fast release ensures lower latency for other agent transfers but potentially wastes DRAM bandwidth. On the other hand, slow release of the bus delivers high bandwidth – assuming the agents are always requesting – but increases the latency for

other agents. The CoreLink DMC-500 scheduler has been designed to manage this bus turnaround effect efficiently. It provides users with programmable parameters and guidelines on when to do this in the most optimized manner, keeping into account the types of agents requesting, the transaction type, coherency, rank, bank and priority.

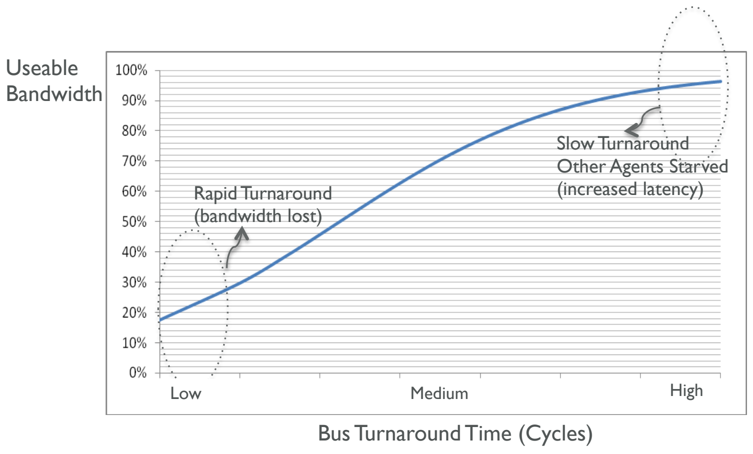


Figure 7: Effect of Bus Turnaround Time on Bandwidth Utilization

6. Queue Fill Threshold – The memory controller has a large internal queue for storing incoming transactions. During operation, it may so happen that all the queue entries are filled with low priority transactions creating back-pressure on subsequent incoming high priority transactions. In order to ensure that these high priority transactions are always serviced, it is necessary to reserve some queue entries for high priority accesses only. The degree of ‘fullness’ of the queue as measured in groups of 1/16th size of the queue is referred to as the “Fill Threshold”. So a Fill Threshold of 16 implies there are no reserved queue entries for subsequent high priority transactions whereas a threshold of 6 implies 62.5% of the queue entries are reserved for high priority transactions. Reserving queue entries reduces the number of opportunities that the scheduler has to optimize the bandwidth utilization and hence results in slightly dropped utilization. Figure 8 shows simulation results achieved for a Fill Threshold of 6. This threshold value was chosen as it only resulted in a 4% loss of bandwidth utilization as seen in the left graph in Figure 8. Lower values of Fill Threshold (implying higher number of reserved queue entries) resulted in a substantial bandwidth loss and would hence not be advisable. The right graph in Figure 8 shows the effect of selected Fill Threshold on the average and maximum latencies for LL and RT transactions. With Fill Threshold of 6 there is a significant reduction in these latencies at only 4% additional loss of bandwidth utilization. This is the type of programming trade-off users of CoreLink DMC-500 can make to optimize the latency response for their LL and RT traffic.

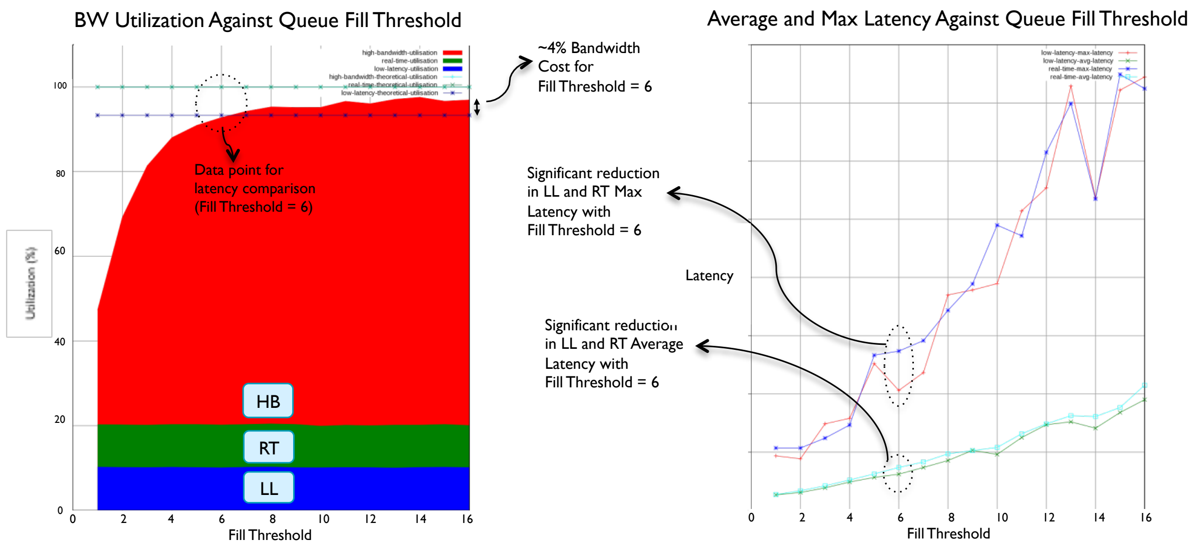
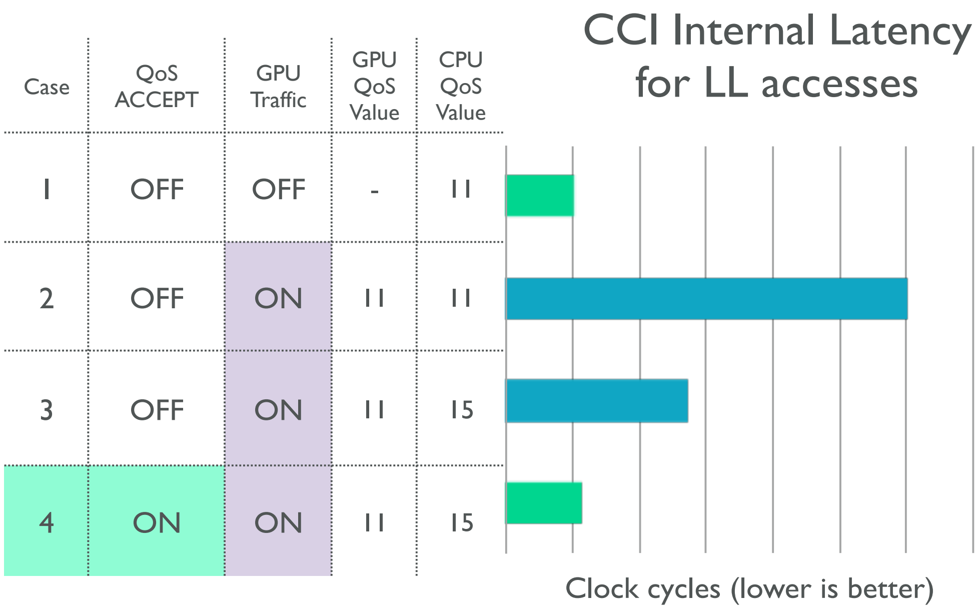


Figure 8: Effect of Queue Fill Threshold on Bandwidth, Average and Max Latencies

7. QoSACCEPT Signaling – All the QoS mechanisms described above operate at the micro-level inside the DMC-500. Super-imposed on these mechanisms are more coarse or macro-level QoS mechanisms that operate in the Corelink CCI-550 interconnect. An in-depth description of these mechanisms is outside the scope of this paper. Our performance analyses of CoreLink CCI-550 and DMC-500 subsystems showed that if the memory controller provides visibility to the interconnect of its queue fullness, the interconnect was able to make more informed decisions at the macro-level, thereby improving system performance significantly. This visibility is provided to the interconnect using QoSACCEPT signaling which indicates the QoS value **above** which the memory controller is willing to accept transactions based on its own internal Queue Fill Threshold and transactions that are in flight. The interconnect then attempts to route transactions with QoS values to the memory controller module that's willing to accept them.

   Figure 9 shows effect of enabling QoSACCEPT in a system with CPU (LL) and GPU (HB) traffic. The memory controller is programmed with a QoS Threshold value of 12, implying that it will back-pressure transactions with QoS value less than 12 if its internal queue crosses a certain Fill Threshold. The graph measures the latency of the transaction within the interconnect before it gets accepted by the memory controller. Consider 4 different cases in this performance experiment:



Case	QoS ACCEPT	GPU Traffic	GPU QoS Value	CPU QoS Value
1	OFF	OFF	-	11
2	OFF	ON	11	11
3	OFF	ON	11	15
4	ON	ON	11	15

Figure 9: Effect of QoSACCEPT on LL access Latency within CCI-550

Case 1 – QoSACCEPT mechanism and GPU traffic are turned off. Since the only traffic flowing through the interconnect is the CPU traffic, there's no back pressure from the memory controller. Consequently the latency of transactions within the interconnect is small.

Case 2 – Next GPU traffic is turned ON at the same QoS value as CPU traffic. Both agents are competing for memory bandwidth and there is significant back-pressure from the memory controller resulting in very long CPU latency within the interconnect.

Case 3 – The CPU QoS value is then increased from 11 to 15 giving it a higher priority over GPU transations. This reduces the latency of CPU traffic within the interconnect.

Case 4 – Finally, keeping the same QoS values for GPU and CPU traffic, QoSACCEPT mechanism is turned ON. When its internal queues start filling up, the memory controller exerts back-pressure on the GPU traffic whose QoS value is lower than the programmed QoSACCEPT threshold. However in this scenario the interconnect exploits the QoSACCEPT signaling to manage traffic flow rather than regulate traffic more generally, and consequently CPU transaction latency drops to almost the same as Case 1.

The QoSACCEPT mechanism is fundamentally the escalation of Queue Fill Threshold information to the memory controller interface. This escalation enables better macro-regulation for the interconnect and increases overall efficacy of the combined solution. This showcases the importance of close integration between the interconnect and memory controller, achieved by CoreLink CCI-550 and DMC-500.

Summary

In this paper, we have explored various memory controller performance optimization techniques for SoCs characterized by mobile style architecture. ARM builds memory controllers for advanced memories targeting mobile SoCs with the goal of providing the best, optimized performance from CPU to memory for LL, HB and RT traffic agents. Although the mobile traffic characteristics described in this paper have been simplified for ease of analysis and explanation, extensive system simulations, hardware emulation using complex, real-world traffic traces and years of partner success confirm that the mobile memory subsystem performance remains true to its design intent. Specifically, integration of interconnect and memory controller QoS provides further improvements in results not achievable by individual blocks alone.

Future work includes extension of this integration to include ARM processors as well, providing a low-latency fast path to memory, performing speculative fetches and optimizing the use of CPU and system caches for further enhancements in memory access performance.