A Bursty Multi-Port Memory Controller with Quality-of-Service Guarantees

Zefu Dai Jianwen Zhu
Department of Electrical and Computer Engineering
University of Toronto, Toronto, ON, Canada, M5S 3G4
{zdai,jzhu}@eecg.utoronto.ca

ABSTRACT

Embedded multimedia system-on-chips place an increasing demand on Multi-Port Memory Controllers (MPMCs) for higher memory system performance and energy efficiency, in addition to satisfying various types of quality-of-service requirements, such as minimum latency and bandwidth guarantees. While previous works have attempted to target different aspects of the MPMC design challenges, none has succeeded in addressing all these problems simultaneously. In this paper, we propose a new approach that can provide, not only minimum latency and bandwidth guarantees, but also higher efficiency in utilization of physical DRAM bandwidth and dynamic bandwidth made available by underutilized ports. Experimental results show that, on typical multimedia workloads, our approach improves the effective DRAM bandwidth and energy efficiency by as much as 1.9x and 1.49x, respectively. In addition, the response latency for latency-sensitive port is improved by more than 10X, while preserving bandwidth guarantee for all ports.

Categories and Subject Descriptors

B.6.1 [**Logic Design**]: Memory Control and Access

General Terms

Algorithms

Keywords

MPMC, QoS, Energy, Bandwidth, Latency, DRAM

1. INTRODUCTION

Modern multimedia System-on-Chips (SoCs) often employ heterogeneous architectures with many subsystems to optimize for power consumption and throughput, and these subsystems all share the same off-chip memory due to cost and pad limits. As a result, the Multi-Port Memory Controller (MPMC), which arbitrates and routes requests and data to and from the external memory, becomes the key component in achieving best overall system performance. Often, there are different types of Quality-of-Service (QoS) requirements from different subsystems. For example, latency sensitive subsystems prefer their requests to be served as early as possible, whereas bandwidth sensitive subsystems require their periodic requests to be serviced within certain period of time. Thus, it becomes mandatory for MPMC to simultaneously satisfy different and often competing QoS requirements.

One challenge is the dynamic nature of requests coming from different ports: it is often difficult to statically predict the bandwidth requirement of each port. A naïve MPMC that statically assigns bandwidth according to a pre-established *service level agreement* (SLA) might not be able to utilize the *residual bandwidth*, made available by those subsystems that, for some period of time, did not make as many requests as permitted by their SLA.

Another challenge is the dynamic nature of available physical bandwidth from DRAM accesses. Contemporary DRAM memory is no longer a truly random access memory that has uniform access time for different access commands. It is a multi-dimensional memory array organized as a 5D structure of $\{channel, rank, bank, row, column\}$. There is nearly an order of magnitude difference in the access latency between consecutive accesses to different columns within the same row and different rows within the same bank. As a consequence, the *effective bandwidth* of DRAM often deviates significantly from its *peak bandwidth*. In general, it is important to avoid *page crossing*, where addresses of two consecutive accesses fall in different rows (or pages).

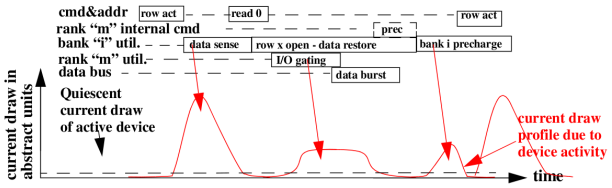

Figure 1: Current profile of a DRAM read cycle.

Page crossing has significant impact on power consumption of battery-powered multimedia system-on-chips such as those found in Mobile Internet Devices (MIDs). DRAM page crossing happens when a pending transaction requires the DRAM to close a previous page and open a new page for read/write operations. Each DRAM page crossing causes the DRAM to issue an activate-prechage command pair which senses tens of thousand of bits into the row buffer and precharge them back into memory cells. As can be seen from Figure 1 [17], the activate-prechage command pair draws much higher current than other DRAM commands and takes a long time to complete. Thus, with a higher page crossing rate, both energy efficiency and effective DRAM bandwidth fall.

Significant progresses have been made in MPMC designs, progressing from heuristics-based QoS-aware controllers, to more advanced QoS-guaranteed controllers. Unfortunately, to the best of our knowledge, none satisfies all requirements

CODES+ISSS'11, October 9–14, 2011, Taipei, Taiwan.

argued earlier. This paper improves upon on our previous MPMC design [4], which enjoys the desired features of QoS-guarantees, but fails to account for the page crossing effect. More specifically, we make the following contributions:

- a new QoS guaranteed scheduling framework that considers the DRAM page crossing effect;
- a new dynamic credit calculation method that best utilizes *residual bandwidth* for latency minimization.

The rest of the paper is organized as follows. In Section 2, we review related work in the literature. In Section 3, we discuss the proposed approach. In Section 4, we describe our evaluation framework. In Section 5, we present and discuss experimental results.

2. RELATED WORK

The importance of MPMC scheduling has led to many efforts, often originated to address different requirements of MPMC designs.

Many early works focus their research on maximizing effective DRAM bandwidth. Rixner *et al.* [16] evaluated the benefits of various heuristics for reordering DRAM memory accesses. They did not identify a conclusive winner among their various heuristics. McKee *et al.* [14] proposed a Stream Memory Controller that combines compile-time stream detection with run-time access order selection. Their approach uses a round robin policy to schedule among multiple stream buffers, and streams as much data as possible from current buffer before going to the next. Hur and Lin [7] developed an Adaptive History-Based memory scheduler for the IBM Power5 processor. The AHB scheduler uses history information to reorder DRAM memory accesses. By encoding different types of history information into FSMs, they provide a mechanism for combining multiple constraints. Their later work [8] extended the AHB scheduler to support simple DRAM power down policy and power throttling. These memory controllers are designed for homogeneous systems where different memory access streams are mixed into the same queues, and there is no distinction of QoS requirements from different applications.

Various approaches have been proposed to design QoS Guaranteed MPMCs at different grain levels, but few of them consciously utilize *residual bandwidth*. Heithecker *et al.* proposed a mixed QoS SDRAM controller for FPGA-based high-end image processing [6]. Their controller uses a priority-based round-robin algorithm to schedule requests from different priority classes. They also implemented a second level bank scheduler to improve DRAM bandwidth by leveraging bank interleaving and request bundling techniques. Subsequent improvements added flow control to the high priority request class to prevent it from over-consuming bandwidth [5]. The improved memory controller was later used in the MORPHEUS reconfigurable architecture [18]. However, their controller only provides bandwidth guarantees at a coarse grain level.

Burchard *et al.*, presented a real-time streaming memory controller (SMC) for mobile device architectures [3]. The SMC aims to provide bandwidth guarantees to support off-chip network services, such as guaranteed real-time data transport. They used a credit-based method to reserve bandwidth for different streams, such that any stream that successfully reserves sufficient credits will be guaranteed to finish its data transport within a predictable period of time. To reduce power consumption, they mandated a full page access as the minimum scheduling slot. The SMC only provides bandwidth guarantees at coarse grain level, and neither residual bandwidth nor latency is taken into consideration.

Sonics' MemMax® Scheduler provides three types of QoS for up to sixteen ports [1]. QoS modes include priority (low latency), allocated bandwidth (guaranteed bandwidth) and best effort (no guarantee). MemMax® filters incoming requests in order of highest cost to lowest cost, where costs are calculated to maximize DRAM efficiency and enforce QoS. Due to its commercial nature, other details of the design are not known.

Lee *et al.* described a QoS-guaranteed MPMC for multimedia platform SoCs [11]. Their controller separates the ports into three different types: latency sensitive, bandwidth sensitive, and don't-care. It uses a credit-based scheme to provide bandwidth guarantees for all types of ports and grants preemptive service to the latency-sensitive clients to minimize latency. However, their bandwidth allocation scheme is fixed at run-time, and the latency-sensitive ports cannot get the full benefit from residual bandwidth. For instance, if a latency-sensitive port exhausts its available credits, it is demoted to the don't-care type until next service cycle.

In our previous work [4], we proposed a MPMC that can provide fine grain QoS guarantees to different QoS classes. This scheduler differs from other works in that, it allocates *residual bandwidth* according to priorities. However, this scheduler fails to account for DRAM bandwidth and energy efficiencies, which this paper attempts to address.

3. PROPOSED APPROACH

A MPMC scheduler is responsible for arbitrating among N ports, each with a queue $i \in [0..N-1]$ holding the requests, and granting access of DRAM memory at each clock cycle. We assume each port has an associated bandwidth guarantee of S_i, where $\Sigma_i S_i \leqslant 1$, for its proportion of total bandwidth. Without loss of generality, we also assume that the ports have descending priority in latency.

This section details our proposed MPMC scheduling algorithm. We start by a baseline scheduling algorithm, the Weighted Round Robin (WRR), analyze its pros and cons, then gradually refine it for better performance.

3.1 Weighted Round Robin

A Weighted Round Robin scheduler serves each queue in a Round Robin manner. For each round of scheduling period with T_{round} number of clock cycles, a *scheduling token* is passed among different queues to determine which port has the right to access DRAM memory. A burst limit BL_i and a burst counter BC_i are associated with each queue to control the length of service for queue i, where $\sum_{i=0}^{N-1} BL_i \leqslant T_{round}$.

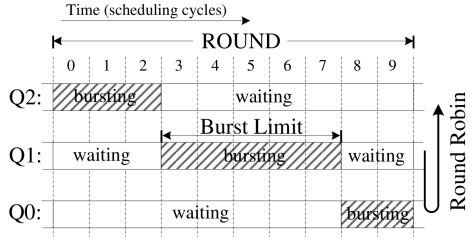

Figure 2: Weighted Round Robin scheduling.

The WRR scheduling is illustrated in Figure 2. Each queue has 2 states in each round of scheduling: Waiting and Bursting. In the Waiting state, the queue waits for the *scheduling token*. And once it gets the token, it enters into Bursting state during which it sends requests to DRAM memory consecutively.

The WRR algorithm guarantees each queue i a minimum bandwidth of BL_i/T_{round} if active, and a bounded waiting time of $(T_{round} - BL_i)$ for each round.

The WRR algorithm achieves bandwidth guarantee and has low hardware implementation cost. It addition, it en-

courages *bursting*, or the continuous serving of requests from the same port, which tends to exhibits address locality within a single row (page). Thus better DRAM bandwidth utilization is naturally achieved by the reduction of page crossing.

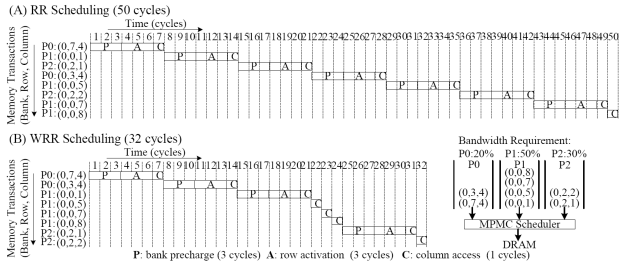

Figure 3: DRAM access result using RR scheduling (A) and WRR scheduling (B).

EXAMPLE 1. *Figure 3 shows the overall latency of scheduling 8 requests from 3 queues using different scheduling algorithms. The access address of each request is represented as a tuple of (bank, row, column). And each memory access cycle may consist of bank precharge, row activation and column access commands depending on the DRAM states. We can see that with a simple Round Robin(RR) scheduler, the 8 requests will take up to 50 cycles to complete. while a WRR scheduler can finish all requests in 32 cycles with less activate-precharge commands, which improves both DRAM performance and energy efficiency.*

The WRR has two drawbacks. First, it cannot provide minimum latency guarantee for latency sensitive queues, since everyone is served in a round robin manner.

Second, it has little control over the use of *residual bandwidth*. During run-time, some queues may become empty or under-utilizing their allocated bandwidth. This only results in shortened T_{round}, while the allocated bandwidth for each queue remains BL_i/T_{round}. For high priority, low bandwidth requirement queues, they will have a small BL_i, and therefore, only enjoy a small portion of benefit from the changing of T_{round}.

3.2 Credit Borrow and Repay

To improve the latency of high priority queues in the WRR scheduler, the Credit Borrow and Repay (CBR) [4] mechanism can be used. The main idea of CBR is: for a specific scheduling algorithm, the minimum latency guarantee can be achieved by allowing higher priority queues to borrow scheduling opportunities from lower priority queues as soon as needed. Borrowed credits are repaid later when the queue in debt accumulates enough residual bandwidth.

When applying CBR to one latency-sensitive queue of the WRR scheduler, a FIFO is needed to record the port from which credit has been borrowed. We call this FIFO the *debtQ*. When the latency-sensitive queue is not empty and the *debtQ* is not full, the CBR borrows the current scheduling opportunity, if it is granted to a lower priority queue, and re-assigns it to the latency-sensitive queue. The port ID of the lending queue is pushed into the *debtQ*. The WRR scheduler continues as if nothing has happened: it does not have to switch the *scheduling token* if it is not planned at current time. The credit borrow process allows the requests in the latency-sensitive queue to be scheduled preemptively, therefore improves the response latency greatly.

To repay the debt, the latency sensitive queue needs to keep a repay credit counter, *repayCredit*, to collect residual bandwidth. Also a debit credit counter $debit_i$ is added to each of the low priority queues. During each round of scheduling, if the latency-sensitive queue does not use up its allocated bandwidth, the residual bandwidth is used by the repay credit counter. In other words, whenever the *repayCredit* is greater than 1, and the *debtQ* is not empty, the CBR pops the *debtQ*, increments the $debit_i$ corresponding to the popped ID, and decrements the *repayCredit*. Low latency queues can then use their debit credits to extend their service time.

The repay process is necessary to provide bandwidth guarantee for the queues being borrowed. The concept of the CBR implemented for the WRR is further illustrated in Example 2.

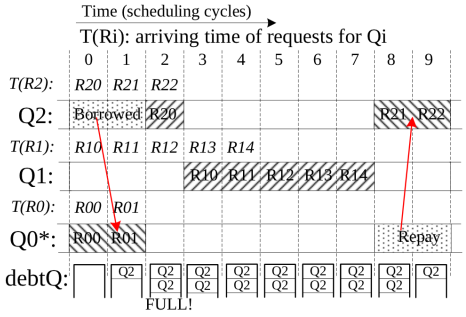

Figure 4: CRB for WRR.

EXAMPLE 2. *Figure 4 illustrates the CBR implemented for Q0 of the WRR scheduler. The first row of numbers denotes the cycle time. And each queue occupies two separate rows for the arriving and scheduling time of its requests, respectively. The bottom row shows the cycle-by-cycle status of the* debtQ. *The CBR mechanism allows Q0 to be served immediately at cycle 0 and 1 by borrowing scheduling opportunities from Q2. Later, when Q0 obtains the* scheduling token*, and there is no request in its queue, it repays the credits to Q2 at cycle 8 and 9. And bandwidth guarantee of the entire system is recovered.*

The advantages of CBR are:

- it allows latency-sensitive queues to be scheduled preemptively, while retaining bandwidth guarantee of the entire system.
- it requires minimal modification to the base WRR scheduling algorithm, since the base scheduler does not need to be aware of the Borrow-and-Repay process.
- it has low implementation cost. The major cost is a FIFO plus a few counters.

We can see from the above example that, with a FIFO depth of 2, the CBR allows 2 requests of Q0 to arrive any time within a T_{round}, while being serviced immediately.

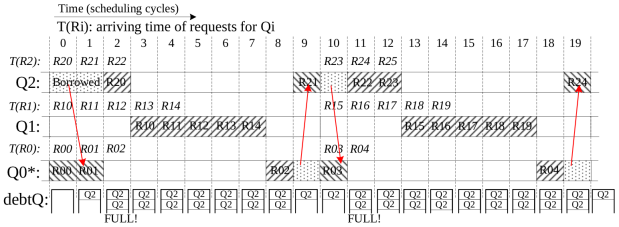

Figure 5: Impact of *debtQ* depth.

However, when 3 requests of Q0 arrives in the first round, the third request will have to wait until cycle 8, when it obtains the *scheduling token* as illustrated in Figure 5. All later requests, for example the *R04* arriving at cycle 11, need to wait for a long time, since the *debtQ* is full, therefore no borrowing can occur. This specific problem may be alleviated by increasing the depth of the *debtQ*. However, for applications that have very un-even distribution of memory access, this solution will quickly become impractical.

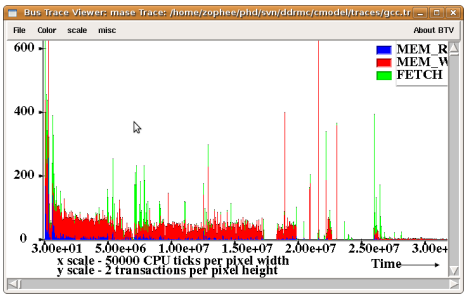

Figure 6: gcc trace.

EXAMPLE 3. *The CPU caches are known to have very uneven distribution of memory accesses. As shown in Figure 6, which describes the memory access requirement over time of a CPU cache when running gcc. Although the average bandwidth requirement over long period of time is very low, its instant bandwidth requirement could be orders of magnitude higher than average in many specific points of time. It will take a large CBR* debtQ *to achieve lowest possible latency for CPU cache over entire process in this case.*

3.3 Residual Bandwidth Calculation

To prevent the *debtQ* from growing too big, the repay credit counter needs to collect residual bandwidth quick enough to reduce the backlog in *debtQ*. One natural idea is, we not only collect residual bandwidth from the queue in debt, but collect it from all queues simultaneously.

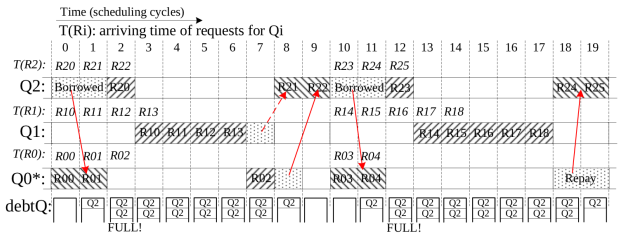

Figure 7: Residual bandwidth from other queue.

EXAMPLE 4. *As shown in Figure 7, suppose Q1 only uses 4 scheduling opportunities in the first round, it will have one unused scheduling slot. If this residual bandwidth is used to help repay the debt in cycle 7, the* debtQ *will become non-full, and all later requests in Q0 including R04 can be scheduled immediately.*

To calculate residual bandwidth, a naïve way treats the unused scheduling slot of queue i in each round as residual bandwidth, which is calculated using $(BL_i - BC_i)$. However, this is not accurate enough because different request arrival patterns with the same request arrival rate could result in different residual bandwidth calculations.

EXAMPLE 5. *Suppose that there are 3 queues, Q0, Q1 and Q2, each has* $BL_i = 1$. *Also assume that* $T_{round} = 3$ *and only 1 request overall arrives in each cycle. For one arrival pattern: R00, R10, R20, R01, R11, R21, . . . , where Rij denotes the jth request of queue i, the WRR scheduling result is: (Q0, Q1, Q2), (Q0, Q1, Q2), . . . , and no residual bandwidth is found. However, for another arrival pattern: R00, R20, R10, R01, R21, R11, . . . , the WRR scheduling order is: (Q0, Q2), (Q1), (Q0, Q2), (Q1), ... Using the naive method, the WRR scheduler will mistakenly think that there is residual bandwidth in every round.*

The above example shows that the WRR scheduler does not have enough information for an accurate residual bandwidth calculation. Therefore, we choose to use the more accurate Bandwidth Guaranteed Prioritized Queueing (BGPQ) algorithm [10], and modify it to work for the WRR scheduler.

3.3.1 Bandwidth Guaranteed Prioritized Queueing

The BGPQ algorithm calculates dynamic bandwidth utilization of each port for every scheduling cycle, and allows the residual bandwidth to be utilized according to the priority of different ports.

Suppose that there are N different ports, ranked $0 > \cdots > N-1$ in terms of priority. Each port has a queue for arriving requests, an active flag A_i, and an empty flag E_i. Each port also has an associated bandwidth guarantee, the static credit S_i, where $\sum_{i=0}^{N-1} S_i \leqslant 1$.

At the beginning of the $(n+1)th$ scheduling cycle, a dynamic credit D_i will be calculated for each queue as follows:

$$D_i(n+1) = D_i(n) + A_i(n) \cdot S_i + k_i(n), i \in \{0, \ldots, N-1\}$$

where

$$A_i(n) = !E_i(n) \tag{1}$$

and

$$k_i(n) = \begin{cases} -1 & \text{if } D_i(n) = max_{j \in \{0,\ldots,N-1\}}\{D_j(n) | A_j(n)\}. \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

The baseline dynamic credit in the current scheduling cycle is equal to the dynamic credit from the previous cycle. Active queues receive additional credit equal to their static credit, whereas non-active queues do not compete for scheduling resources and their dynamic credits remain unchanged. As the scheduler grants the non-empty queue with the highest D_i access to the back-end memory resources, that queue's dynamic credit is decremented by 1 to reflect that it used a quantum of its available bandwidth.

When $\sum_{i=0}^{N-1} A_i(n) \cdot S_i < 1$, there is residual unused bandwidth to be distributed according to the priorities of the different queues. Thus we re-allocate the residual bandwidth to the non-empty queue with the highest priority by adding a corresponding amount of credit to its dynamic credit. The total residual bandwidth is:

$$d(n) = 1 - \sum_{i=0}^{N-1} A_i(n) \cdot S_i \tag{3}$$

The dynamic credit calculation equation is then augmented to include the residual bandwidth allocation:

$$\begin{aligned} D_i(n+1) \quad = \quad & D_i(n) + A_i(n) \cdot S_i + k_i(n) \\ & + r_i(n), i \in \{0, \ldots, N-1\} \end{aligned} \tag{4}$$

where

$$r_i(n) = \begin{cases} d(n) & \text{if } i = min_{j \in \{0,\ldots,N-1\}}\{j | A_j(n)\}. \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

Compared to many other bandwidth guaranteed scheduling algorithms, the BGPQ algorithm has the following desirable characteristics:

- it provides prioritized access to the residual bandwidth available at run-time instead of sharing it among all request queues. By having control over the residual bandwidth, we can better utilize the limited resources to enhance system performance.
- it has low complexity which facilitates efficient hardware implementation and introduces little latency to the scheduling data path.

3.3.2 New Dynamic Credit Calculation System

The BGPQ algorithm is designed to always schedule the queue with maximum dynamic credit in each cycle. Since the WRR uses a very different scheduling scheme, the concept of Dynamic Credit Calculation in the BGPQ algorithm needs to be modified before applying to the WRR.

Same as the BGPQ algorithm, we use dynamic credit to represent the bandwidth utilization status of each queue. Active queues get a static credit in each cycle, while the queue being granted in the previous cycle needs to minus 1 from its dynamic credit as indicated by Equation 4. But we need to change Equation 2 first, since the WRR uses a different scheduling scheme. Now the queue that gets the *scheduling token* in the previous cycle needs to deduct 1 from its dynamic credit to reflect the usage of its allocated bandwidth. And the equation becomes:

$$k_i(n) = \begin{cases} -1 & \text{if } schToken(n) = i. \\ 0 & \text{otherwise.} \end{cases} \quad (6)$$

A second change is made to Equation 1, which now becomes:

$$A_i(n) = (D_i(n) < 0) \;||\; (!E_i(n) \& D_i(n) < M_i) \quad (7)$$

where $M_i = (T_{round} - BL_i) \cdot S_i$ is the upper limit for the dynamic credit of each queue. This upper limit is derived from the following two facts: first, in any T_{round} of time, a busy queue i will accumulate a maximum of $T_{round} \cdot S_i$ dynamic credit and consume them all during its BL_i length of Bursting state, therefore, $T_{round} \cdot S_i = BL_i$. Second, the dynamic credit of queue i only grows during its Waiting state (it needs to be deducted by 1 for each bursting cycle), which has a maximum length of $T_{round} - BL_i$. Therefore, if queue i fully utilizes its allocated bandwidth, the maximum dynamic credit it can achieve is $M_i = (T_{round} - BL_i) \cdot S_i$. When it does not generate as many requests as permitted, it should accumulate less than M_i of dynamic credit in each round.

The active flag is modified to capture residual bandwidth in the WRR algorithm. This equation aims to account for the following two cases:

- Queue i may enter into Bursting state before it has accumulated enough dynamic credit for all bursting requests, resulting in a negative dynamic credit when it exits the Bursting state. This means it has consumed its future share of bandwidth to complete request bursting. Therefore, it should stay active to claim allocated bandwidth as long as its dynamic credit is negative.

- non-empty queue i can accumulate credits up to the limit of M_i. Dynamic credits after the maximum limit should become residual bandwidth automatically.

The first case is relatively easy to understand since the future bandwidth that has already been booked cannot be collected as residual bandwidth. The second case is important to capture residual bandwidth due to under-utilization, compared with residual bandwidth due to empty queue case. This is because:

When queue i is not empty and has less requests than allowed, it may need to wait for a maximum of $T_{round} - BL_i$ cycles before getting scheduled in each round. During the waiting time, its dynamic credit will keep increasing since queue i is not empty, and therefore 'active' if using Equation 1. Then, because it does not have enough requests to consume all accumulated credit during the Bursting state, a positive dynamic credit will be carried forward to the next round. If this case keeps going on, its dynamic credit will grow without an upper bound. While with Equation 7, when queue i is under-utilizing its bandwidth, it will quickly reach the upper limit and become inactive, even thought the queue is not empty. The under-utilized bandwidth will then become residual bandwidth.

A third change is required to account for the credit repay process: if a queue obtains the *scheduling token* and its debit credit counter is non-zero, it does not need to deduct 1 from its dynamic credit in the next cycle. Instead, it only needs to deduct 1 from its debit counter, which means it uses repaid credit to send one request to the DRAM. Therefore, we modify Equation 6 to the following:

$$k_i(n) = \begin{cases} -1 & \text{if } (schToken(n) = i \;\&\; debit_i(n) = 0). \\ 0 & \text{otherwise.} \end{cases} \quad (8)$$

Also, Equation 5 needs to be modified to reflect that residual bandwidth can be used by non-latency-sensitive queues only when the *debtQ* is empty:

$$r_i(n) = \begin{cases} d(n) & \text{if } (i = min_{j \in \{0,\ldots,N-1\}}\{j | A_j(n)\} \\ & \;\&\; debtQ.empty()). \\ 0 & \text{otherwise.} \end{cases} \quad (9)$$

With the above modifications, residual bandwidth of the WRR scheduler can be correctly calculated using Equation 3.

Algorithm 3.1: The Proposed Scheduling Algorithm

Input: request queues $\{Q_i\}$ and static credits $\{S_i\}$.
Output: Scheduled queue id, GrantID.

```
   /* Credit Repay Process                                          */
 1 Update dynamic credits D_i and residual bandwidth d using
   Equations(3, 4, 7, 8, 9);
 2 if !debtQ.empty() then
 3     repayCredit ← repayCredit + d;
 4     if repayCredit ⩾ 1 then
 5         repayID ← debtQ.pop();
 6         debit [repayID] ← debit [repayID] + 1;
 7         repayCredit ← repayCredit - 1;

   /* WRR Scheduling                                                */
 8 if BC[schToken ] ⩾ BL[schToken ] or Q[schToken ].empty() then
 9     BC[schToken ] ← 0;
10     schToken ← find next non-empty queue using Round Robin;
11 else if debit [schToken ] > 0 then
12     debit [schToken ] ← debit [schToken ] - 1;
13 else
14     BC[schToken ] ← BC[schToken ] + 1 ;

   /* Credit Borrow Process                                         */
15 if !Q[0].empty() and schToken ≠ 0 and !debtQ.full() then
16     GrantID = 0;
17     debtQ.push(schToken);
18 else
19     GrantID = schToken;
20 return GrantID;
```

Finally, the entire algorithm can be more accurately described by Algorithm 3.1, suppose Q0 is the highest priority queue implemented with the CBR mechanism.

4. EVALUATION METHODOLOGY

In this section, we describe the methodology to evaluate the strength and weakness of the proposed scheduling algorithm.

4.1 Benchmarks

We target multimedia embedded SoCs, which usually consist of multiple subsystems like CPU, accelerators, display and so on. We refer to the set of memory transactions issued by the subsystems as *workloads*. In this work, we simulate 2 different types of workloads: CPU workload generated by running cache simulation on standard SPEC2000 integer benchmarks; and multimedia workload generated by running ALPBench [12] suite of parallel multimedia applications.

We chose to directly use the CPU trace file provided in [9]. Figure 6 visualizes the trace file for the gcc benchmark. The horizontal axis captures time in CPU instructions executed

and the vertical axis captures the number of DRAM transactions after the cache. Being a typical CPU application that features rich temporal and spatial locality, the memory demand of the benchmark is as low as around 20MB/s. Equally typical, the burstness of the trace is irregular. We conclude that this is suitably representative of a typical MPMC user with high priority, low bandwidth requirement and highly dynamic, random access pattern.

The ALPBench [12] suite of parallel multimedia applications was selected as the source for accelerator memory trace data as we believe these applications and their access patterns are representative of multimedia hardware accelerators typically embedded in SoCs. As shown in Table 1, these applications consist of MPEG2 video decoding (MPGdec) and encoding (MPGenc); image processing in the form of face recognition (Face_Rec); audio processing in the form of voice recognition (Sphinx); and graphics processing in the form of ray tracing (Ray_Trace). These are non-trivial applications rather than mere kernels, as evidenced by the lines-of-code reported in the second column.

Column 3 and 4 of Table 1 show the size of captured trace files, as well as the corresponding bandwidth requirement. Figure 8 visualizes the trace file captured for MPGenc. It is evident that it has much higher bandwidth requirement than the gcc trace. It also features a periodically repeated access pattern. Like the other traces from the ALPBench, it is a typical stream application with high bandwidth demand and regular access pattern.

Table 1: Summary of benchmarks.

Benchmark	# LOC	Trace Size(MB)	Bandwidth Requirement (GB/S)
Face_Rec	19,741	2.7	1.41
MPGenc	13,313	70.3	1.02
MPGdec	15,321	7.9	1.04
Ray_Trace	12,597	683.9	1.22
Sphinx	29,317	173.9	0.89

All benchmarks in ALPBench are multi-threaded. As such, they provide us an opportunity to emulate accelerator behaviour by assuming each thread's work maps to a separate hardware accelerator. Thus we can capture the accelerator workload by capturing separate memory trace file for each working thread. Also need to mention that, we leverage the Pin binary instrumentation framework [13] to generate the trace files for the ALPBench applications.

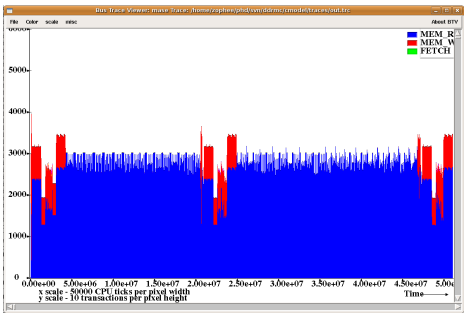

Figure 8: MPG-enc trace.

4.2 Simulation Framework

To evaluate the performance and energy efficiency of the proposed MPMC, an accurate simulation model of DRAM memory and PHY is needed. We use the DRAMSim2 [9], which can cycle-accurately simulate the latency of arbitrary DRAM command sequence for specific DRAM devices using the parameters provided by vendors. The DRAMSim2 also implements the Micron DDR2 Power Model [15] to facilitate power consumption evaluation. It has simple interface to support different memory scheduling algorithms. The entire simulation framework is depicted in Figure 9, trace files from different workloads are fed into a cycle-accurate C model of the proposed MPMC, which then drives the DRAMSim2. We use the Micron 32M_4B_x4_sg3E DDR2 device, provided in the DRAMSim2 package, as our DRAM memory model.

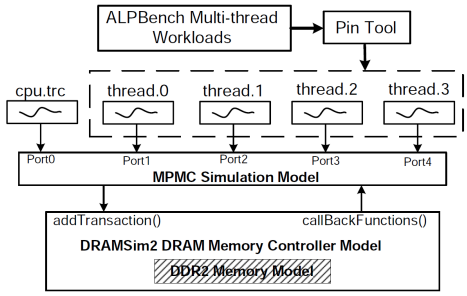

Figure 9: Simulation Framework.

5. EXPERIMENTS

5.1 Experimental Setup

We simulate a 5-port MPMC using the platform shown in Figure 9. Port 0 has the highest priority and port 4 the lowest. The CPU trace uses port 0 and the ALPBench traces occupy four ports from port 1 to port 4 for their four different threads. The initial static credits of different ports are set as follows: $S_0 = 2\%$, $S_1 = 30\%$, $S_2 = S_3 = S_4 = 20\%$. We make $\sum_{i=0}^{N-1} S_i = 92\%$, so there will be 8% of residual bandwidth at run-time. Given the DRAM system configuration and the bandwidth requirement of the ALPbench suite, port 1 to port 4 have much higher bandwidth demand than allocated, thus little residual bandwidth can be expected from these ports. The CBR is implemented for port 0 only and its depth is set to 16.

To make the simulation results more understandable, we chose 4 other scheduling algorithms for comparison: Priority Queueing(PQ), Round Robin(RR), Bandwidth Guaranteed Prioritized Queueing(BGPQ) and the original Weighted Round Robin(WRR). We ran the trace for 10 million cycles to get stable outputs, and collected the results of effective DRAM bandwidth, DRAM power consumption, average DRAM transaction latency and bandwidth allocation.

We measure the latency of memory transactions in the following way: for write requests, the latency is the time from a write request being presented at the MPMC user port to when the data is written into DRAM memory; for read requests, the latency is the time from a read request being presented at the MPMC user port to the return of the read data. The average latency for each port is obtained by dividing the sum of all latencies with the total number of memory transactions.

5.2 Experimental Results

As described in section 3, our proposed scheduling algorithm can support bursting scheduling on a granularity defined by T_{round}. With a larger T_{round}, each port can send more requests to the DRAM memory consecutively every time it gets the *scheduling token*. It is therefore interesting to evaluate the performance of the proposed scheduler with different lengths of T_{round}, measured in terms of number of scheduling cycles.

Table 2: Bandwidth(GB/S) VS round length(cycle).

PERIOD	10	50	100	200
Face_Rec	1.319	1.897	2.008	2.060

Table 2 shows the effective DRAM bandwidth achieved by the proposed scheduler with different lengths of T_{round}

when running the workload of Face_Rec. When the T_{round} increases from 10 to 100, the DRAM performance is improved greatly. However, DRAM bandwidth is not improved as much when the T_{round} is further increased from 100 to 200. This is because DRAM memory has a typical page size of 2 KByte, which supports maximal 64 consecutive transactions, each with a data size of 32 Bytes (64bit x 4). With $T_{round} = 100$, each port could exploit the open page policy of DRAM to the maximum if it has small memory access stride. To get a better result of our scheduler, we set the T_{round} to 100 in all later experiments.

Table 3: DRAM bandwidth and power consumption.

	FaceRec	MPGenc	MPGdec	RayTr	Sphinx
Average Effective DRAM Bandwidth(GB/S)					
PQ	1.909	1.418	1.690	1.626	1.760
BGPQ	1.333	1.257	1.538	1.231	0.997
RR	1.449	1.517	1.510	1.203	0.806
WRR	2.022	1.519	1.679	1.986	1.948
PROP*	2.008	1.514	1.674	1.974	1.928
Average DRAM Power Consumption(Watts)					
PQ	4.311	3.798	3.917	3.928	3.940
BGPQ	4.257	4.171	4.324	4.105	3.822
RR	4.276	4.499	4.398	4.116	3.536
WRR	4.277	3.875	3.818	4.220	3.914
PROP*	4.277	3.879	3.821	4.217	3.903
Average Act-Pre Power Consumption(Watts)					
PQ	1.169	1.007	0.950	1.005	0.956
BGPQ	1.495	1.471	1.439	1.421	1.319
RR	1.437	1.623	1.529	1.452	1.169
WRR	1.064	1.017	0.863	1.048	0.820
PROP*	1.073	1.025	0.870	1.052	0.823

*PROP = Proposed Scheduler

Table 3 shows the effective DRAM bandwidth and power consumption of different schedulers. The proposed scheduler achieves almost the same DRAM bandwidth and power consumption as the WRR scheduler in all cases. Compared to the BGPQ and RR scheduler, the WRR and our proposed scheduler achieve 1.1x to 1.9x better effective DRAM bandwidth, and have less or comparable power consumption. This proves that the WRR and our proposed scheduler enhance the DRAM performance by reducing the page crossing rate, as is evident by much less activate-precharge power consumption shown in the bottom group of the table.

Table 4: Energy Efficiency.

	PQ	BGPQ	RR	WRR	PROP*
GB/S per Watt	0.42	0.306	0.308	0.454	0.452
Act-Pre Ratio(%)	25.6	34.6	34.6	23.9	24.1

*PROP = Proposed Scheduler

Table 4 shows the achieved bandwidth per watt and activate-precharge power consumption ratio of different schedulers. The proposed scheduler achieves 49% better energy efficiency and its activate-precharge power consumption ratio is reduced to 24.1%, compared to 34.6% of the RR scheduler.

Table 5: Bandwidth Allocation for Face_Rec.

Port	0	1	2	3	4
PQ	0.9%	83.9%	15.0%	0.0%	0.0%
BGPQ	1.4%	38.6%	20.0%	20.0%	20.0%
RR	1.3%	24.7%	24.7%	24.7%	24.7%
WRR	0.9%	33.0%	22.0%	22.0%	22.0%
PROP*	0.9%	33.1%	22.0%	22.0%	22.0%

*PROP = Proposed Scheduler

Although the PQ scheduler achieves almost the same effective DRAM bandwidth and energy efficiency as the proposed scheduler, its scheduling results are not desirable since it can cause starvation in low priority ports. This can be proved by the bandwidth allocation results for the Face_Rec workload as shown in Table 5. It can be seen from the table that port 3 and 4 are starved in the PQ scheduler. The RR scheduler divides the available DRAM bandwidth evenly among all busy ports. And the BPGQ, WRR and the proposed scheduler allocate the bandwidth according to their static credit configuration. However, the BGPQ algorithm re-allocates the residual bandwidth completely to the non-empty port with highest priority. The WRR scheduler re-allocates the residual bandwidth according to the static credit configuration. In comparison, our proposed scheduler firstly utilizes all residual bandwidth to improve the response latency of port 0, then shares the rest of residual bandwidth among all non-latency-sensitive ports according to their static credits, as is explained in the next experiment.

Table 6: Average CPU response latency (ns).

	FaceRec	MPGenc	MPGdec	RayTr	Sphinx
PQ	64.95	85.26	72.54	71.88	73.14
BGPQ	313.5	300.4	262.3	314.3	416.3
RR	128.5	115.8	122.2	134.73	206.9
WRR	697.0	956.5	844.6	700.6	705.7
PROP	63.64	79.85	74.37	68.17	70.39
W/P*	11.0x	12.0x	11.4x	10.3x	10.0x

*W/P=WRR_Latency/PROPOSED_Latency

We compare the average response latency for the CPU workload in Table 6. It shows that the proposed scheduler achieves comparable response latency in all cases as the PQ scheduler. The latency achieved by the PQ scheduler can serve as the lower bound since the requests from CPU workload are scheduled immediately on their arrivals. Therefore, it proofs that the proposed scheduler can provide minimum latency guarantee from the front-end scheduler point of view. The WRR scheduler performs badly in the CPU response latency, as the CPU has to wait for up to 98% of T_{round} before being served in each round. Although the BGPQ scheduler can utilize the residual bandwidth to improve the CPU's response latency, the CPU still needs to compete with other ports for DRAM access. While in our proposed scheduler, the CBR mechanism combined with the new Dynamic Credit Calculation System allows the CPU requests to be executed preemptively and results in an over 10x improvement in the response latency, compared to the original WRR scheduler.

Table 7: Latency (ns) without residual bandwidth.

	FaceRec	MPGenc	MPGdec	RayTr	Sphinx
PROP*	302.2	230.8	473.0	348.1	337.0
pref.	-4.7x	-2.9x	-6.4x	-5.1x	-4.8x

*PROP = Proposed Scheduler

Finally, we examine the impact of residual bandwidth on the response latency. We reset the static credit of each port to the following: $S_0 = 2\%$, $S_1 = 30\%$, $S_2 = 28\%, S_3 = S_4 = 20\%$. Now we have $\sum_{i=0}^{N-1} S_i = 100\%$, and little residual bandwidth can be found at run-time. The resulting CPU's response latency is shown in Table 7: Without utilizing the residual bandwidth, the CPU may see an average of 5x drop in performance.

5.3 Hardware Cost

We implemented an 8-port MPMC on a Xilinx Virtex-6 XC6VLX240T-1 device found on Xilinx's ML605 evaluation board. The depth of the CBR *debtQ* is set to 16 and the T_{round} is set to 100. Implementation and analysis were performed using version 13.1 of the Xilinx ISE Design Suite.

The top level diagram of the MPMC design is shown in Figure 10. A MPMC scheduler is deployed between user request queues and a back-end DDR PHY. Logically, one *request queue* is placed in front of each MPMC port. The queue implementation can be chosen based on user requirements, ranging from a simple collection of registers to a complete FIFO. The MPMC scheduler implements the schedul-

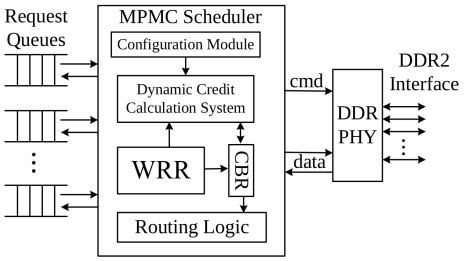

Figure 10: Top level diagram of MPMC.

ing algorithm described in Section 3. It consists of a Run-Time Configuration module, a Weighted Round-Robin Scheduler, a Dynamic Credit Calculation System, a Credit Borrow and Repay system as well as a data routing logic.

The Run-Time Configuration Interface allows the user to dynamically configure the bandwidth allocation and priority of each port at run-time. The Weighted Round-Robin Scheduler is implemented using the priority rotation method: once a port gets the scheduling token, its priority drops to the lowest, and the priority of the other ports are updated according to their distance to the scheduled port. The token is passed to the next non-empty port with highest priority when current port exhausts its scheduling credits.

The Dynamic Credit Calculation System implements Equation (3, 4, 7, 8, 9) as described above. Equation 3 and 9 rely on states of all queues, thus could become critical paths easily. All the other equations rely on local information of each queue only and can be parallelized efficiently. The Credit Borrow and Repay system requires only standard FIFOs and counters. More implementation details of the Dynamic Credit Calculation and Credit Borrow and Repay system can be found in our previous work [4]. Finally, the routing logic is a crossbar controlled by the scheduler.

Table 8: Performance and resource utilization.

	LUT	FF	BRAM	Max_Freq
Xilinx	3450	5540	1-9	200
Proposed	1393	884	0	167
Speedy	1986	1380	4	198

Table 8 shows the hardware cost of both Xilinx's and our proposed MPMC design. In [19], Xilinx gives estimated resource utilization and maximum clock frequency of their MPMC design, using virtex6 -1 device and ISE 13 tool. Xilinx's MPMC implements Round-Robin scheduling algorithm by default and supports a maximum of 8 ports. Their reported resource utilization accounts for the PHY, arbiter, control logic and data path. In comparison, our proposed 8-port MPMC design costs only 1393 LUTs and 884 registers, and achieves a maximum clock frequency of 167MHz. If combined with the Speedy DDR PHY designed by Ray Bittner [2], the total cost of the design is still smaller than Xilinx's MPMC design.

6. CONCLUSION

Our study concludes that the proposed approach successfully addresses the multi-port memory controller design requirements of providing minimum latency and bandwidth guarantees for different QoS classes while maximizing effective DRAM bandwidth and energy efficiency. By combining three techniques into one simple algorithm, our approach improves both DRAM bandwidth and energy efficiency by as much as 1.9x and 1.49x, respectively. And the cache response latency is improved by more than 10x while retaining minimum bandwidth guarantee for all ports. Moreover, the proposed design has low hardware cost. It costs less than 1.4K LUTS on a Virtex-6 FPGA, and requires mostly simple logic that can be efficiently parallelized.

7. ACKNOWLEDEGEMENTS

The authors like to thank the support of National Sciences and Engineering Research Council of Canada, as well as China Scholarship Council for the first author.

8. REFERENCES

[1] MemMax® scheduler. Datasheet, February 2010.

[2] R. Bittner. The speedy DDR2 controller for FPGAs. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems & Algorithms*, Las Vegas, USA, July 2009.

[3] A. Burchard, E. H. Nowacka, and A. Chauhan. A real-time streaming memory controller. In *Proceedings of the Design Automation and Test in Europe Conference and Exhibition*, Munich, Germany, 2005.

[4] Z. Dai, M. Jarvin, and J. Zhu. Credit borrow and repay: Sharing DRAM with minimum latency and bandwidth guarantees. In *ICCAD'10: Proceedings of the 2010 IEEE/ACM International Conference on Computer-Aidded Design*, San Jose, CA, USA, 2010.

[5] S. Heithecker and R. Ernst. Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements. In *Proceedings of the Design Automation Conference*, Anaheim, California, USA, June 2005.

[6] S. Heithecker, A. C. Lucas, and R. Ernst. A mixed QoS SDRAM controller for FPGA-based high-end image processing. In *Workshop on Signal Processing Systems Design and Implementation*, August 2003.

[7] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *Proceedings of the 37th International Symposium on Microarchitecture*, Oregon, Portland, 2004.

[8] I. Hur and C. Lin. A comprehensive approach to DRAM power management. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, Salt Lake, UT, USA, 2008.

[9] B. Jacob. DRAMSim2. http://www.ece.umd.edu/dramsim/, April 2010.

[10] K. L. E. Law. The bandwidth guaranteed prioritized queuing and its implementation. In *Proceedings of the Global Telecommunications Conference*, 1997.

[11] K. B. Lee, T. C. Lin, and C. W. Jen. An efficient quality-aware memory controller for multimedia platform SoC. *IEEE Transactions on Circuits and Systems for Video Technology*, 15(5), May 2005.

[12] M. L. Li, R. Sasanka, S. Adve, Y. K. Chen, and E. Debes. The ALPBench benchmark suite. In *Proceedings of the IEEE International Workload Characterization Symposium*, pages 34–45, October 2005.

[13] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI'05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2005.

[14] S. A. Mckee, W. A. Wulf, J. H. Aylor, M. H. Salinas, R. H. Klenke, S. I. Hong, and D. A. B. Weikle. Dynamic access ordering for streamed computations. In *IEEE Transaction on Computers*, 2000.

[15] Micron Technology, Inc. Calculating memory system power for DDR2, 2006.

[16] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, Canada, 2000.

[17] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. Dramsim: A memory system simulator. In *SIGARCH Computer Architecture News*, 2005.

[18] S. Whitty and R. Ernst. A bandwidth optimized SDRAM controller for the MORPHEUS reconfigurable architecture. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, April 2008.

[19] Xilinx, Inc. LogicCORE IP multi-port memory controller (v6.04.a)., 2011.