Credit Borrow and Repay: Sharing DRAM with Minimum Latency and Bandwidth Guarantees

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

ABSTRACT

Multi-port memory controllers (MPMC) play an important role in system-on-chips by coordinating accesses from different subsystems to shared DRAMs. The main challenge of MPMC design is optimize quality-of-service by simultaneously satisfying different—and often competing—requirements, including bandwidth and latency. While previous works have attempted to address the challenge, the proposed solutions are heuristic and often cannot provide bandwidth and/or latency guarantees. In this paper, we propose a new technique called Credit-Borrow-and-Repay (CBR) that augments a dynamic scheduling algorithm drawn from the networking community, improving it to achieve minimum latency while preserving minimum bandwidth guarantees. Our experiments show that on typical multimedia workloads, the cache response latency can be improved as much as 2.5X.

Categories and Subject Descriptors

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

General Terms

Alogrithm, Design, Experiment, Performance

Keywords

Multiport Memory Controller, Bandwidth Guarantee, Minimum Latency, Credit Borrow and Repay

1. INTRODUCTION

The overall system performance of modern multimedia System-on-Chips (SoCs) is increasingly limited by memory system performance due to growing disparities between logic and memory speeds. This trend is worsening as DRAM bandwidth improvements lag system bandwidth requirements for new applications. For example, dual channel H.264 has a bandwidth requirement of over 10GB/s [6] whereas a single DDR2-800 memory module has a theoretical peak bandwidth of 6.4GB/s. Adding to the demand for raw DRAM bandwidth is the requirement of efficient sharing: media SoCs often employ heterogeneous architectures with many subsystems to optimize for power consumption and performance, and these subsystems all share the same off-chip memory due to cost and pad limitations. To coordinate the competing accesses, a Multi-Port Memory Controller (MPMC) arbitrates and routes requests and data to and from the external memory.

Example 1. *Figure 1 depicts a real example of a portable media player SoC in which several subsystems are integrated, including an embedded central processing unit (CPU) running a rich operating system and audio processing, a video accelerator for H.264 video decoding, a Digital-Visual-Interface (DVI) controller for driving a display, and an 8-port MPMC. Each subsystem contains one or more ports to the MPMC, which provides access to the DDR2 SDRAM. Note that the CPU subsystem uses two ports for its instruction and data caches. The video accelerator has five ports for its intra-prediction, inter-prediction, sum, and deblocking filter (DF) modules. The DVI controller has one port to access the frame buffer. The bandwidth requirements of all ports are detailed in Table 1.*

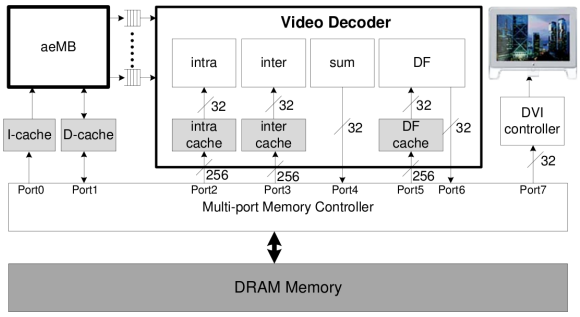

Figure 1: A portable media player SoC.

The challenge of MPMC design is threefold:

1. The MPMC must simultaneously satisfy a diverse set of bandwidth requirements from all ports. In Example 1, the bandwidth requirement can differ as much as 1000X.

2. The MPMC has to simultaneously satisfy latency requirements without violating 1. In Example 1, ports 0 and 1—which connect to the I-cache and D-cache of the CPU subsystem—must be serviced as early as possible, even though their bandwidth requirement is relatively low.

3. The MPMC must adapt to the dynamic nature of the application workload. In Example 1, the traffic generated on port 0 and 1 are cache misses by unknown applications. The traffic generated by the H.264 decoder might change given different video clips, and will

Table 1: Bandwidth requirements of portable media player System-On-Chip.

Port	Module	Bandwidth (MB/s)
0	I-cache	6.4
1	D-cache	9.6
2	intra	1.175
3	inter	164.77
4	sum	0.09
5	DF	31.00
6	DF	156.70
7	DVI	94.00

definitely increase when decoding fast-moving action movies as opposed to romance movies.

To quantify the figure of merit for MPMC design, the notion of **Quality-of-Service (QoS)** has been borrowed from the telecommunications community, which has dealt with similar issues for decades. More specifically, we can formulate MPMC design targeting QoS as:

- Optimize for **minimum latency**, represented as $\Sigma_i W_i L_i$, where L_i is the sum of time it takes the MPMC to service each individual memory transactions on port i and W_i is a weight reflecting the priority or relative importance of latency on port i;
- Subject to **guaranteed bandwidth** $\forall i, BW_i \geq \overline{BW}_i$, where BW_i is the number of transactions serviced per unit time for port i and $\overline{BW}_i$ is the minimum bandwidth requirement on port i.

EXAMPLE 2. *For Example 1, we can set* $W_0 = W_1 = 1$, *and* $W_2 = W_3 = W_4 = W_5 = W_6 = W_7 = 0$. *We can also derive* $\overline{BW}_i$ *from Table 1.*

Guaranteed bandwidth is usually achieved by employing algorithms that establish fairness between different users. Round Robin and Weighted Fair Queue are two examples. Latency can be improved by allowing high priority requests to execute preemptively. However, it is not feasible to achieve both fairness and prioritized access by simply combining these algorithms together. Moreover, to accommodate dynamic workloads, the MPMC should be able to efficiently utilize the *residual bandwidth*: *i.e.*, the bandwidth assigned to one user that is unused at a specific point of time.

While a large body of research has explored efficient MPMC designs, few provide high-quality resolutions for both QoS classes concurrently. For those that can provide guarantees for both bandwidth and latency [11], the bandwidth allocation is fixed and can not adapt dynamically to changes of traffic at runtime; consequently, the controller has little control over the utilization of available residual bandwidth and the latency suffers.

In this paper, we propose a dynamic MPMC scheduling algorithm that can satisfy both classes of QoS requirement. More specifically, we make the following contributions:

- We show that a dynamic scheduling algorithm [10] originally proposed in the networking community can be adapted to solve the MPMC problem;
- We quantify the hardware implementation cost of the algorithm;
- We enhance the algorithm such that it not only guarantees bandwidth, but also achieves minimum latency.

As a result, our MPMC can achieve minimal latency for latency sensitive users while enforcing bandwidth guarantees for all users. By updating its bandwidth allocation ratio at each scheduling cycle, the proposed scheduler adapts well to unpredictable and/or changing memory traffic patterns.

Table 2: Previous work.

	Bandwidth	Bandwidth+Latency
QoS-aware	[8]	[7] [2]
QoS-guaranteed	[5]	[11]

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 give implementation details. In Section 5, we describe our evaluation framework. In Section 6, we present and discuss experimental results.

2. RELATED WORK

The importance of MPMC scheduling has led to the publication of many approaches. These approaches vary their objective with respect to QoS, and can be categorized as follows (see Table 2):

QoS-aware where bandwidth or latency is heuristically improved;

QoS-guaranteed where minimum bandwidth is guaranteed and/or minimum latency is achieved;

Heithecker *et al.* proposed a mixed QoS SDRAM controller for FPGA-based high-end image processing [8]. Their controller uses a priority-based round-robin algorithm to schedule requests from different priority classes. However, their algorithm cannot provide bandwidth guarantees to low-priority classes. Subsequent improvements add flow control to the high priority request class to prevent it from consuming too much bandwidth [7]. The improved memory controller is later used in the MORPHEUS reconfigurable architecture [15]. The flow control unit improved the fairness of their algorithm; however, bandwidth guarantees are still not enforced for priority classes.

Burchard *et al.*, present a real-time streaming memory controller (SMC) for mobile device architectures [5]. The SMC aims to provide bandwidth guarantees to support off-chip network services, such as guaranteed real-time data transport. They use 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 optimize for power consumption, they mandate a full page access as the minimum scheduling slot. The SMC only provides bandwidth guarantees at coarse grain level and latency is not taken into consideration.

Ackermann *et al.* employ a memory management abstraction for self-reconfigurable video processing platforms [2]. Their controller uses a priority queue scheduling algorithm and relies on design automation software running on PC to specify QoS requirements. The software collects the QoS requirements of the entire platform in advance and configures the SDRAM controller accordingly to ensure that there is no unsatisfiable requirement. The effectiveness of the resultant controller is limited by the accuracy of the designer's predictions of the operating conditions.

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.* describe a QoS-aware memory controller 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 port and grants preemptive service to the latency-sensitive clients to minimize latency. However, their bandwidth allocation scheme is fixed at runtime, and the latency-sensitive ports cannot 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.

3. MAIN IDEAS

A MPMC scheduling algorithm is the method by which the data flows are given access to the DRAM bandwidth. In the case where only one DRAM channel is considered, the scheduler simply arbitrates among multiple data flows, granting access to the one with the highest score. The problem of managing QoS becomes one of scoring each data flow to meet bandwidth guarantees for every data flow and to minimize response latency for high priority data flows.

While many algorithms endeavour to provide bandwidth guarantees, this work employs the Bandwidth Guaranteed Prioritized Queueing (BGPQ) approach [10]. One desirable characteristic of BGPQ is that it provides prioritized access to the residual bandwidth available at runtime instead of sharing it among all data flows. By having control over the residual bandwidth, we can better utilize the limited resources to enhance system performance. Another strength of BGPQ is its low complexity, facilitating efficient hardware implementation that introduces little latency to the data processing pipeline. BGPQ falls short in that it cannot achieve the lowest possible latency for users with both high priority and low bandwidth requirements. This work addresses that deficiency by implementing a Credit-Borrow-and-Repay (CBR) mechanism to improve the BGPQ algorithm so that it can achieve the minimum latency for high priority users while retaining all its original features.

3.1 The BGPQ Algorithm

This section briefly introduces the Bandwidth Guarantee Priority Queuing (BGPQ) algorithm. The fundamental objectives of BGPQ are to provide minimum bandwidth guarantees to different QoS classes and to allow the residual unused bandwidth to be utilized according to the priority levels of different classes.

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

At the beginning of each scheduling cycle, a dynamic credit D_i will be calculated for each class as follows:

$$D_i(n+1) = D_i(n) + \overline{E_i(n)} \cdot S_i + k_i(n), i \in \{1, \ldots, N\} \quad (1)$$

where

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

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

When $\sum_{i=1}^{N} \overline{E_i(n)} \cdot S_i < 1$, there is residual unused bandwidth to be distributed according to the priorities of the different QoS classes. 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 $d(n) = 1 - \sum_{i=1}^{N} \overline{E_i(n)} \cdot S_i$. The dynamic credit calculation equation is then augmented to include the residual bandwidth allocation:

$$\begin{aligned} D_i(n+1) &= D_i(n) + \overline{E_i(n)} \cdot S_i + k_i(n) \\ &\quad + r_i(n), i \in \{1, \ldots, N\} \end{aligned} \quad (2)$$

where

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

The BGPQ algorithm dynamically determines the bandwidth ratio of each request class at every scheduling slot, making it flexible and efficient in memory bandwidth utilization. The advantages of BGPQ can be illustrated by the examples in Table 3, taken from [10]. Suppose there are three different classes, $1 > 2 > 3$ in priority, with corresponding static credits $S_1 = 50\%$, $S_2 = 30\%$ and $S_3 = 20\%$. From these examples, we can find that when the system is overloaded, the BGPQ algorithm can stop the high priority classes from consuming too much bandwidth and can provide minimum bandwidth guarantees to the lower priority classes. The BGPQ does not fairly allocate all residual resources to all classes, and the residual resources will be allocated to higher priority classes with preference.

Table 3: Examples of BGPQ resources allocation ($S_1 = 50\%$, $S_2 = 30\%$ and $S_3 = 20\%$).

	Input class bandwidth(%)			Allocated class bandwidth(%)		
	1	2	3	1	2	3
1	100	0	70	80	0	20
2	80	60	0	70	30	0
3	30	70	100	30	50	20
4	10	60	30	10	60	30

In term of complexity, the BGPQ requires at most three additions and one comparison to calculate the dynamic credit for each queue. It also needs another comparison to decide which queue has the highest dynamic credit. Hence, the scheduling can be done efficiently without adding much latency to the entire pipeline.

3.2 The Credit Borrow and Repay Mechanism

Although BGPQ can provide bandwidth guarantees and prioritized access to residual bandwidth, it is not good at minimizing latency for latency-sensitive classes with low bandwidth requirements.

EXAMPLE 3. *For example, suppose there are 3 request classes—CPU, Video Decoder and DVI Controller—with minimum bandwidth guarantees of 20%, 50% and 30%, and priority of 1, 2 and 3 respectively. Assuming all queues are non-empty for enough time, the results of BGPQ credit calculation and scheduling are shown in Table 4.*

Table 4 illustrates that although the CPU has the highest priority, its requests are scheduled on cycles 2 and 7. However, this is plainly suboptimal in terms of CPU latency as the schedule can be rearranged with CPU requests in cycles 0 and 1 while providing the same guaranteed bandwidth for the other classes, as shown in Table 5. The service for latency-sensitive high priority classes can be improved by allowing them to borrow the scheduling slots of lower priority classes, provided they return the scheduling opportunities later when they have accumulated enough dynamic credit.

Based on this observation, we propose the Credit Borrow and Repay (CBR) mechanism to improve the BGPQ

Table 4: The BGPQ scheduling sequence.

Time	0	1	2	3	4	5	6	7	8	9
Grant	V	D	C	V	V	D	V	C	D	V
Dynamic Credit Transition of Different Classes										
CPU	0.2	0.4	0.6	-0.2	0	0.2	0.4	0.6	-0.2	0
Video	0.5	0	0.5	1	0.5	0	0.5	0	0.5	1
DVI	0.3	0.6	-0.1	0.2	0.5	0.8	0.1	0.4	0.7	0

*C:CPU V:Video D:DVI

algorithm. The CBR can be implemented by adding a *debt counter* and a *debt queue* for each high-priority class (CPU in this example). The debt counter keeps track of the number of slots CPU has borrowed and the debt queue records the IDs of whom it borrowed from. When the CPU queue is non-empty and its debt counter does not exceed a maximum limit, CPU will borrow the current scheduling slot if that slot would otherwise be assigned to a lower priority class. If borrowing occurs, the debt counter is incremented and ID of the class selected by BGPQ will be enqueued in the debt queue. When the debt counter reaches the maximum limit or the CPU queue is empty, the debt will be repaid by transferring dynamic credits from the CPU to the class IDs in the debt queue.

Table 5: Optimized scheduling sequence.

Time	0	1	2	3	4	5	6	7	8	9
Grant	***C***	***C***	***V***	V	V	D	V	***D***	D	V
Dynamic Credit Transition of Different Classes										
CPU	0.2	0.4	0.6	-0.2	0	0.2	0.4	0.6	-0.2	0
Video	0.5	0	0.5	1	0.5	0	0.5	0	0.5	1
DVI	0.3	0.6	-0.1	0.2	0.5	0.8	0.1	0.4	0.7	0

*C:CPU V:Video D:DVI

We modify the dynamic credit calculation equation to incorporate our CBR mechanism. Now the equation becomes:

$$D_i(n+1) = D_i(n) + \overline{E_i(n)} \cdot S_i + k_i(n) + r_i(n) + debt_i(n) + payment_i(n), i \in \{1, \ldots, N\} \quad (3)$$

where

$$debt_i(n) = \begin{cases} -1 & \text{if queue } i \text{ has to repay the debt.} \\ 0 & \text{otherwise.} \end{cases}$$

and

$$payment_i(n) = \begin{cases} 1 & \text{if queue } i \text{ got paid from another port.} \\ 0 & \text{otherwise.} \end{cases}$$

In the above equation, $debt_i(n)$ indicates whether queue i have to pay the debt at cycle n and $payment_i(n)$ indicates if queue i got paid from another queue at cycle n. The complete algorithm is shown in Algorithm 3.1.

3.3 The CBR Configuration

Because the CBR only transfers credits from one port to another, it does not break the balance of the BGPQ credit system and all the advantages of BGPQ are retained. The relationship between BGPQ and CBR can be illustrated by Figure 2.

While BGPQ serves to enforce bandwidth guarantees across scheduling cycles, CBR allows the scheduler to act as a priority queue within a period of L scheduling cycles. The period L is determined by the depth of the debt queue as well as the bandwidth allocation. Suppose queue i has a bandwidth guarantee of S_i and a debt queue depth of M_i. When its debt queue is full, it needs to repay all the debts; the total debt is bounded by M_i, the maximum number of scheduling slots borrowed from the other queues. In order to repay all the debts, queue i would require at most $L = M_i/S_i$ scheduling cycles to accumulate enough dynamic credits. By configuring S_i and M_i, we are able to control the granularity of bandwidth guarantee, and the burstiness of each port—that

Algorithm 3.1: The CBR Scheduling

Input: Request valid flag $reqValid(i)$ for each queue i.
Output: Scheduled queue id, GrantID.

```
1  foreach queue i do
2      Calculate dynamic credit D_i(n+1) using Equation 3.;
3      if !reqValid(i) or full(debtQ(i)) then
4          flag(i) ← FALSE;
5      else
6          flag(i) ← TRUE;
7  GrantID = max_{i∈[flag(i)==TRUE]} D_i(n+1);
8  Borrowed ← FALSE;
9  BorrowID ← 0 ;
10 foreach queue i from high priority to low priority do
11     if GrantID!= i and !Borrowed and !full(debtQ(i)) and
       reqValid(i) then
12         push(debtQ(i),GrantID);
13         BorrowID ← i;
14         Borrowed ← TRUE;
15     else if full(debtQ(i)) or (!empty(debtQ(i)) and
       !reqValid(i)) and D_i(n+1) >= 1 then
16         payment(pop(debtQ(i))← TRUE;
17         debt(i) ← TRUE;
18 if Borrowed then
19     return BorrowID;
20 else
21     return GrantID;
```

is, each port can have a maximum burstiness of M_i at a distance of M_i/S_i. This feature can help us to optimize the response latency for high priority queue. Also, by allowing requests from the same queue to execute in a bursty way, it helps to improve the back-end memory efficiency when an open page policy is employed. This is particularly useful for stream applications, which often feature consecutive memory access pattern.

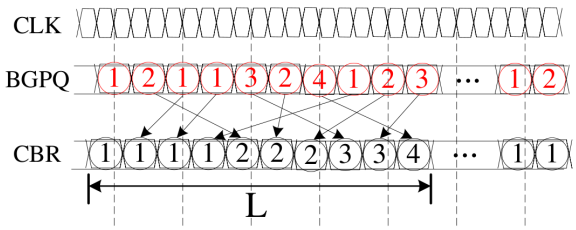

Figure 2: Relationship of BGPQ and CBR.

4. IMPLEMENTATION

To evaluate the efficiency of our proposed CBR algorithm, we implemented a Multi-Port Memory Controller (MPMC) in an FPGA. The number of ports is instantiation-time configurable and each port has a run-time programmable priority and static credit. The top level diagram of our MPMC is shown in Figure 3.

Logically, one *request queue* is placed in front of each MPMC port. The queue implementation can be chosen based on class requirements, ranging from a simple collection of registers to a complete FIFO. The MPMC itself is composed of a *request scheduler* and a *back-end DDR PHY*. These two components are connected by a *command path* and a *data path*.

4.1 Implementation Details

The request scheduler controls the access to the back-end PHY. It consists of a *BGPQ credit assigner*, a *sorting network* and a *CBR module*. The BGPQ credit assigner calculates a dynamic credit for each queue at every scheduling

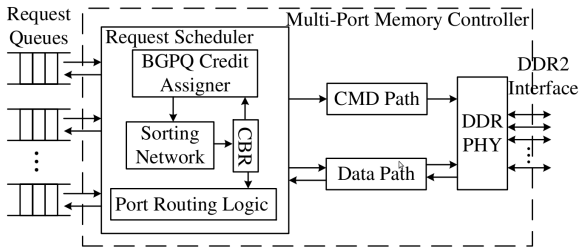

Figure 3: Top level diagram of MPMC.

cycle. The calculation, described by Equation 3, includes at most three additions and one comparison. The circuit for this dynamic credit calculation is shown in Figure 4. If a queue is not empty, its dynamic credit D_i will be incremented by its static credit S_i. Then, if the queue was granted in the previous scheduling cycle, its dynamic credit is decremented by one. Finally, if the queue has the highest priority, we will add a residual credit to its dynamic credit. The comparison is needed to determine which queue has the highest priority. The residual bandwidth is calculated only once at every scheduling cycle and distributed to all ports.

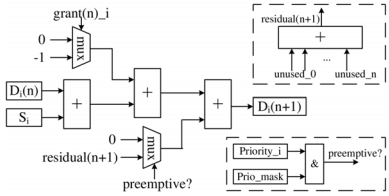

Figure 4: Dynamic credit calculation circuit.

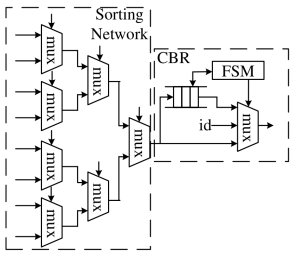

Figure 5: Top level diagram of MPMC.

The sorting network is responsible for comparing all the dynamic credits and reports which port has the highest dynamic credit in current scheduling cycle. The sorting network is implemented as a simple tree of comparators controlling multiplexers.

The CBR module decides if its associated port should borrow the current scheduling slot. If yes, it will replace the ID selected by the sorting network with its own queue ID and stores the ID selected by the sorting network into the debt queue. Otherwise, if one queue has debts to repay, the CBR module will enforce the queue to transfer a fixed amount of dynamic credits to the queue with the ID popped from the debt queue. A picture of the CBR together with sorting network logic is shown in Figure 5.

The command path module forwards requests from request scheduler to the back-end DDR PHY. The data path forwards the write data that comes along with write requests to the DDR PHY and returns the read data from DDR PHY to the corresponding ports.

The back-end DDR PHY translates each request into a series of DDR commands and drives the signals into the off-chip DDR interface. In our design, we used the Speedy DDR2 Controller designed for Xilinx Virtex-5 FPGAs by Ray Bittner [4]. The Speedy DDR2 Controller is a compact and fast DDR2 controller that supports an open page operation for a maximum of 4 banks, which can greatly enhance the performance of read and write accesses to the open pages.

4.2 Implementation Cost

We implemented an 8-port MPMC with BGPQ scheduler and CBR module on a Xilinx Virtex-5 XC5VLX50T FPGA, as found on the Xilinx Virtex-5 LXT ML505 Evaluation Platform [17]. Implementation and analysis was performed using version 12.1 of the Xilinx ISE Design Suite [16].

The sorting network requires a three-level tree of comparators controlling multiplexers to determine the winner at each scheduling slot. The multiplexers need to switch among eight commands and their associated write data, resulting in a significant demand for routing that accounts for the bulk of the delay in the scheduling pipeline. To ameliorate the clock frequency, we divided the scheduling process into a two-stage pipeline; with this optimization, the entire design successfully routes with a 150MHz clock period constraint.

Table 6: Details of device utilization and power consumption; CBR(l), CBR(b), and CBR(f) implement a 16-entry debt queue in LUTRAM, BRAM, and slice registers, respectively.

	Slice LUTs	**Slice Registers**	**Block RAMs**	**Power (W)**
Speedy	1986	1380	4	3.311
BGPQ	1129	812	0	0.011
BGPQ+CBR(b)	1251	826	1	0.011
BGPQ+CBR(l)	1277	834	0	0.009
BGPQ+CBR(f)	1436	962	0	0.018
TOTAL(b)	3237	2206	5	3.322
TOTAL(l)	3263	2342	4	3.320
TOTAL(f)	3422	2342	4	3.329

As Table 6 shows, the CBR component with a 16-entry debt queue incurs an area overhead of 10% to 27% relative to the baseline BGPQ module. The CBR component's area is dominated by the debt queue, a structure that grows linearly with its depth. However, the debt queue may be implemented in one of several ways, depending on constraints such as resource availability in a given design instance: if BRAMs are plentiful, they provide an opportunity to implement depth queues as deep as 4096 entries for eight ports without incurring a prohibitive area cost; if BRAMs are scarce, LUTRAMs may be used instead. If the application requires that the debt queue be initialized on the assertion of a reset signal, slice registers must be used and a significantly greater area penalty is incurred.

Table 6 also demonstrates the power overhead of the CBR module to be negligible relative to the other power sinks in the system. The Speedy DDR controller alone provides a baseline power consumption that includes major draws due to IOBs (2.322 W), PLLs (0.227 W), and leakage (0.601 W). In particular, the IOB power consumption is very high due to the large number of DDR pins and the use of digitally controlled impedance (DCI) [3]: the 105 pins of the DDR2 interface account for 2.284 W. The variable portion of the power cost is dominantly attributable to clock distribution and BRAMs. The BRAM-based debt queue implementation uses the fewest clocked resources but requires slightly more power than the LUTRAM implementation due to the additional BRAM. The slice register implementation consumes the most power due to the large number of active slices and the significant additional clock distribution required.

5. EVALUATION METHODOLOGY

In this section, we describe the methodology to evaluate the strengths and weaknesses of the proposed CBR algorithm.

5.1 Simulation Framework

We target multimedia applications. A typical media application processor, as shown in Example 1 includes subsystems such as CPU, accelerators and display, each of which uses one or more ports to access the DRAM. We refer to the set of memory transactions issued by the subsystems as *workloads*. Workload characteristics can be highly dynamic in nature, varying with the type of subsystem and the application running on it.

Validating new system research requires carefully crafting experiments from which credible conclusion can be drawn. This includes the careful choice of a diverse set of workloads. To benchmark scheduling algorithm for MPMC, three approaches can be used to generate workloads.

One approach is to use the *statistical approach* where "traffic" is generated randomly according to a specified distribution. While this approach has been widely used, it is questionable if the simple distribution functions employed by the networking community can sufficiently capture the dynamics of SoC subsystem traffic.

Second, the *execution-based approach* requires the time-domain simulation of the entire system. This could be carried out either at the register-transfer level or at the architectural level. For the latter, one typically needs to use an instruction set simulator to simulates the CPU applications simultaneously with C models of the accelerators, display, and MPMC. The cost of full-system simulation is prohibitively high even at the architectural level for the purpose of MPMC algorithm evaluation. Moreover, it is difficult to obtain C models for the accelerators.

We instead adopt the *trace-based approach* to create reproducible, representative, and comprehensive workloads, while avoiding the prohibitive cost of full-system implementation for all applications. In this method, the CPU, accelerator, and display workloads are separately obtained in the form of memory trace files, each capturing the time-stamped memory transactions of a port.

As depicted in Figure 6, these trace files are used to drive a cycle-accurate C model of the proposed MPMC couple with a simple C model of DRAM. The MPMC model implements both the original BGPQ algorithm, as well as the proposed CBR algorithm.

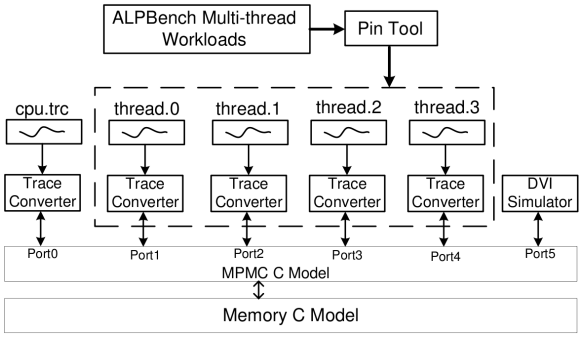

Figure 6: Simulation Framework.

5.2 CPU Workload

Modern embedded processors use caches to reduce DRAM access bandwidth. It is therefore necessary to obtain a CPU workload through cache simulation. We chose to directly use the trace files generated by running cache simulation on standard SPEC2000 integer benchmarks, using freely available traces from [9]. While it is debatable whether one would run any SPEC2000 benchmark while watching a video or playing a game, they are generally accepted as representative of integer applications running on CPUs, and their availability and reproducibility outweigh their disadvantages.

Figure 7 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 spacial locality, the memory demand of the benchmark is as low as about 0.4 memory references per thousand instructions. Equally typical, the burstiness 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 patterns.

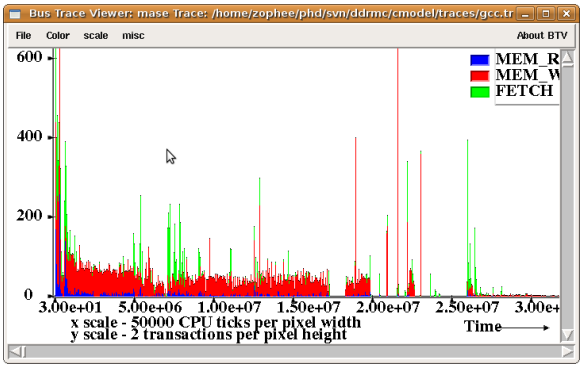

Figure 7: gcc trace.

5.3 Accelerator Workload

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 7, These applications consist of MPEG2 video encoding (MPGenc) and decoding (MPGdec); 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). The lines-of-code reported in the second column is evidence to the complexity of these applications.

Table 7: Summary of benchmarks.

Benchmark	# LOC	Trace Size (MB)	Bandwidth Requirement (txns per 1K instr)
Face_Rec	19,741	2.7	52.1
MPGdec	15,321	7.9	19.4
MPGenc	13,313	70.3	18.3
Sphinx	29,317	173.9	14.3
Ray_Trace	12,597	683.9	26.84

All benchmarks in ALPBench are multithreaded. As such, they provides us an opportunity to emulate accelerator behavior by assuming each thread's work maps to a separate hardware accelerator. Thus we can capture the accelerator workload by capturing separate memory trace files for each working thread.

The procedure for generating memory traces leverages the Pin binary instrumentation framework [13] to modify precompiled programs at runtime by analyzing them at the subroutine, basic block, or instruction level and inserting calls to instrumentation routines. Our instrumentation code is first executed at thread creation, initializing a thread-specific trace file. A thread-specific instruction counter is created and each instruction is instrumented to trigger an increment of that counter on execution. Memory-access instructions are instrumented to emit to the thread's trace file

the accessed address, the type of access (read/write), and the number of instructions executed since the last memory access in that thread. This information is sufficient for the simulator to correctly model bank, rank, row, and command switching delays in the back-end DDR controller while providing an approximation of the amount of computation between memory accesses. Command-line switches provide the ability to restrict tracing to a particular subroutine.

Memory tracing was enabled only during the execution of the main worker thread function, thereby eliminating lead-in and lead-out and providing results representative of steady-state operation. All five applications in the suite were compiled as directed in the ALPBench documentation with two exceptions: all Makefiles were adjusted to use four concurrent threads, yielding four traces of concurrent memory accesses per application; and 'noinline' directives were added to four functions across three benchmarks (MPGenc, Face_Rec, and Sphinx) in order to provide detectable entry and exit points for those functions, as required for the correct operation of the tracing instrumentation.

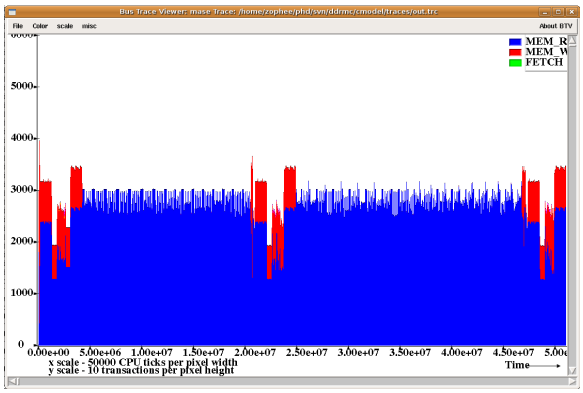

Figure 8: MPG-enc trace.

Columns 3 and 4 of Table 7 show the size of captured trace files, as well as the corresponding bandwidth requirement in number of transactions per thousand instructions. Figure 8 shows the trace file captured for MPGenc. It is evident that it has much higher bandwidth requirement, about 100 times higher than 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.

5.4 Display Workload

The workload for the DVI display controller is trivial to generate: since it linearly scans the frame buffer periodically, we only need to emulate this periodic behavior given the bandwidth requirement of a given resolution. We chose the 1080p resolution for the experiment.

6. RESULTS

The simulation platform is shown in Figure 6. Six ports of the MPMC are used. Port 0 has the highest priority and port 5 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. Finally, port 5 is assigned to the DVI controller.

According to the features of different traces, we configured the static credits of different ports as follows: $S_0 = 1\%$, $S_1 = S_2 = S_3 = S_4 = 20\%$, and $S_5 = 19\%$. We simulate CBR for port 0 only.

The DRAM bandwidth is difficult to measure, particularly when an open page policy and bank interleaving technique are used. The bandwidth greatly depends on the address sequence of the requests. To simplify the evaluation, our memory model simulates a closed page policy in which each transaction has a fixed processing time. This allows us to isolate the impact of the back-end memory behaviour and focus on the analyzing of our scheduler's performance. According to the DRAM specification from Micron [14], a DDR2-533 module takes approximately 55 ns for a read/write cycle, which is 7 clock cycles with a 125MHz clock. The memory controller adds an overhead of about 3 cycles. Therefore, we make our memory model capable of processing one transaction every 10 cycles assuming a 125MHz clock frequency. The maximum bandwidth is 100 transaction per 1000 cycles. We ran one billion cycles for each benchmark to collect the average bandwidth and latency data for each port.

Table 8: Results for the BGPQ scheduler.

Port	0	1	2	3	4	5
Face_Rec						
Latency	32.50	34.28	41.31	52.19	57.67	58.18
Bandwidth	0.4	26.13	22.53	18.46	16.90	15.56
MPGdec						
Latency	11.44	15.51	16.75	19.09	21.45	34.26
Bandwidth	0.4	18.61	18.54	18.02	17.49	15.56
MPGenc						
Latency	13.76	19.18	20.66	22.58	25.26	32.80
Bandwidth	0.4	17.90	17.84	17.49	17.27	15.56
Sphinx						
Latency	10.39	9.38	9.70	10.51	10.88	23.68
Bandwidth	0.4	14.49	14.38	14.51	14.24	15.56
Ray_trace						
Latency	12.51	22.74	25.38	29.13	33.45	38.63
Bandwidth	0.4	20.82	20.04	18.99	17.88	15.56

For port 0 to port 4, we use a read blocking policy, where each read access will block subsequent memory transactions until the read data comes back. For port 5, non-blocking policy is used for all the read requests.

The latency for each memory transaction is calculated as follows: for write requests, the latency is the time from a write request being presented at the MPMC user port to its acceptance by the MPMC; 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.

In the trace files, the distance between two memory transactions is defined by instruction counts. However, in simulation, we need to know when to issue the memory transaction to the MPMC. We translate each instruction distance into a fixed number of clock cycle latency.

Table 8 shows the latency and bandwidth of each port of the BGPQ scheduler for each benchmark. In cases where the MPMC is overloaded, the BGPQ scheduler provides a minimum bandwidth guarantee for every port. For example, both the Face_Rec and Ray_Trace benchmarks exceed the maximum system bandwidth of 100 transaction per 1000 cycles. Even so, the lowest priority port—port 5—still has its bandwidth requirement satisfied. Although port 3 and port 4 achieve less than their 20% guaranteed bandwidth, it is due to the read blocking policy that stops multiple requests being pipelined and causes extra processing overhead. For each benchmark, higher priority ports achieve better bandwidth and latency, since they have higher priority to utilize the dynamic residual bandwidth at run time. For example, port 1 is 70% faster in latency and 50% better in bandwidth compared to port 4 in benchmark Face_Race. Another observation is that although port 0 has the highest priority, it does not necessarily achieve the best latency. This is because it is assigned only 1% guaranteed bandwidth, which means it will need to wait 100 $sheduling\ cycle*$ 10 $cycle\ per\ transaction$ = 1000 $clock\ cycles$ for its turn to use the DRAM memory when there is no residual bandwidth available. The latency is improved greatly by the residual bandwidth allocation mechanism. However, this la-

tency gets worse as the contention between different ports increases.

Table 9: Results for the CBR scheduler.

Port	0	1	2	3	4	5
Face_Rec						
Latency	18.27	34.27	41.30	52.17	57.66	58.18
Bandwidth	0.4	26.13	22.53	18.46	16.90	15.56
MPGdec						
Latency	6.56	15.53	16.79	19.13	21.46	34.20
Bandwidth	0.4	18.60	18.53	18.01	17.49	15.56
MPGenc						
Latency	7.56	19.17	20.64	22.56	25.26	32.78
Bandwidth	0.4	17.90	17.83	17.49	17.27	15.56
Sphinx						
Latency	8.36	9.37	9.73	10.53	10.88	23.68
Bandwidth	0.4	14.49	14.38	14.50	14.24	15.56
Ray_trace						
Latency	5.94	22.74	25.37	29.10	33.43	38.63
Bandwidth	0.4	20.82	20.04	18.99	17.89	15.56

Table 9 shows the results for the CBR scheduler with a debt queue depth of 16. The latency of port 0 is improved by about 1.8x with negligible impact on the other ports. As analyzed in Section 3.3, a debt queue of 16 allows port 0 to burst up to 16 requests and repay the credit in at most $16/1\% * 10 = 16,000$ cycles if no residual bandwidth is available. In the ideal case, if the CPU requests are distributed evenly across the run time, it will take 40,000 cycles to produce 16 requests, which is within the worse-case capability of the 16-deep debt queue. However, the results show that the bandwidth requirement of the CPU trace varies significantly over time; as such, the optimal debt queue depth also varies with time.

Table 10: Impact of debt queue size.

BenchMarks	BGPQ	CBR				PQ
		4	16	64	256	
Face_Rec	32.50	19.71	18.27	14.48	7.64	5.61
MPGdec	11.44	7.57	6.56	5.62	5.62	5.61
MPGenc	13.76	8.61	7.56	6.33	5.61	5.61
Sphinx	10.39	8.67	8.36	7.54	6.33	5.61
Ray_trace	12.51	7.27	5.94	5.68	5.68	5.61
Improvement	1.0	1.54	1.74	2.0	2.52	2.87

To better understand the impact of debt queue size on the performance, extra experiments were carried out with different debt queue depths, the results of which are shown in Table 10. The first column of the table gives the result of the BGPQ scheduler. The last column presents the result of a naive priority queue (PQ) scheduler, which completely ignores the bandwidth requirement and schedules transaction according to a predetermined priority (in this case in descending order of the ports). The PQ scheduler provides a lower bound on the achievable latency on the CPU port.

We can find that as the debt queue size increases, the latency of CPU port keeps improving. The amount of improvement varies across different benchmarks due to their varying residual bandwidth availability. For example, the Ray_Trace benchmark requires a depth of 16 to near the optimal latency whereas Face_Rec will require a depth of more than 256. The system implementer must evaluate the tradeoff between performance and chip area based on the characteristics of the application. In theory, a sufficiently deep debt queue will achieve the lower bound latency on the CPU port.

7. CONCLUSION

Our study concludes that Credit-Borrow-and-Repay (CBR) is an effective approach to addressing the dual constraints of bandwidth guarantees and latency minimization in Multi-Port Memory Controllers (MPMCs). This augmentation of the Bandwidth Guaranteed Priority Queuing (BGPQ) algorithm borrowed from the networking community is highly effective, achieving on-average 2.5X better latency than BGPQ, while maintaining the required bandwidth guarantees. Moreover, the CBR hardware is small: the cost of the entire scheduler is no more than 1.3K 6-LUTs on an FPGA, with the CBR extension circuitry as small as 122 LUTs. We expect its successful use in a wide range of media applications.

8. ACKNOWLEDEGEMENTS

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

9. REFERENCES

[1] MemMax® scheduler. `http://www.sonicsinc.com/uploads/pdfs/memmaxscheduler_DS_021610.pdf`. Datasheet, February 2010.

[2] K. F. Ackermann, B. Hoffmann, L. S. Indrusiak, and M. Glesner. Providing memory management abstraction for self-reconfigurable video processing platforms. *International Journal of Reconfigurable Computing*, 2009.

[3] D. Banas. Using digitally controlled impedance: Signal integrity vs. power dissipation considerations. `http://www.xilinx.com/support/documentation/application\_notes/xapp863.pdf`.

[4] 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.

[5] 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.

[6] P. Casini. SoC architecture to multichannel memory management using Sonics IMT. `http://www.sonicsinc.com/uploads/pdfs/Multichhanel%20White%20Paper%207%202%2008.pdf`. White Paper, July 2008.

[7] 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.

[8] 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.

[9] B. Jacob. DRAM trace files. `http://www.ece.umd.edu/dramsim/`.

[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 âĂŹ05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2005. ACM.

[14] Micron, Inc. DDR2 SDRAM. `http://www.micron.com/products/dram/ddr\_sdram.html`.

[15] 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.

[16] Xilinx, Inc. Design tools. `http://www.xilinx.com/tools/designtools.htm`.

[17] Xilinx, Inc. Virtex-5 LXT FPGA ML505 evaluation platform. `http://www.xilinx.com/products/devkits/HW-V5-ML505-UNI-G.htm`.