A Single-Tier Virtual Queuing Memory Controller Architecture for Heterogeneous MPSoCs

Journal:	*Transactions on Design Automation of Electronic Systems*
Manuscript ID	TODAES-2016-P-1351
Manuscript Type:	Paper
Date Submitted by the Author:	11-Aug-2016
Complete List of Authors:	Song, Yang; University of California San Diego, Electrical and Computer Engineering Samadi, Kambiz; Qualcomm Research Lin, Bill; University of California, San Diego, Electrical and Computer Engineering
Computing Classification Systems :	Heterogeneous (hybrid) systems

SCHOLARONE™
Manuscripts

Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

A

A Single-Tier Virtual Queuing Memory Controller Architecture for Heterogeneous MPSoCs

YANG SONG, University of California San Diego
KAMBIZ SAMADI, Qualcomm Research
BILL LIN, University of California San Diego

Heterogeneous MPSoCs typically integrate diverse cores, including application CPUs, GPUs, and HD coders. These cores commonly share an off-chip memory to save cost and energy, but their memory accesses often interfere with each other, leading to undesirable consequences like a slowdown of application performance or a failure to sustain real-time performance. The memory controller plays a central role in meeting the QoS needs of real-time cores while maximizing the CPU performance. Previous QoS-aware memory controllers are based on a classic two-tier queuing architecture that buffers memory transactions at the first tier, followed by a second tier that buffers translated DRAM commands. In these designs, QoS-aware policies are used to schedule competing transactions at the first stage, but the translated DRAM commands are serviced in FIFO order at the second stage. Unfortunately, once the scheduled transactions have been forwarded to the command stage, newly arriving transactions that may be more critical cannot be serviced ahead of those translated commands that are already queued at the second stage. To address this, we propose a scalable memory controller architecture based on Single-Tier Virtual Queuing (STVQ) that maintains a single-tier of request queues and employs an efficacious scheduler that considers both QoS requirements and DRAM bank states. In comparison with previous QoS-aware memory controllers, the proposed STVQ memory controller reduces CPU slowdown by up to 13.9% while satisfying all frame rate requirements. We propose further optimizations that can significantly increase row-buffer hits by up to 66.2% and reduce memory latency by up to 19.8%.

CCS Concepts: •**Computer systems organization** → *Heterogeneous (hybrid) systems;*

Additional Key Words and Phrases: memory controller, heterogeneous systems, QoS memory scheduling

ACM Reference Format:
Yang Song, Kambiz Samadi, and Bill Lin, 2016. A Single-Tier Virtual Queuing Memory Controller Architecture for Heterogeneous MPSoCs. *ACM Trans. Des. Autom. Electron. Syst.* V, N, Article A (January YYYY), 24 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

For Peer Review

1. INTRODUCTION

Modern heterogeneous MPSoCs [Qualcomm 2015; NVIDIA 2015] have been widely deployed in mobile devices to reduce power while improving system performance. These MPSoCs typically integrate a diverse collection of cores, including real-time cores like GPUs, HD video coders, and display engines, as well as general-purpose cores like CPUs for running applications. To save cost and energy, these cores commonly share resources, among which, the sharing of the off-chip memory is one of the most challenging because memory performance often has a direct and substantial impact on system

This work was supported by a Qualcomm Fellow-Mentor-Advisor (FMA) award.
This work was presented in part at the 53rd Annual ACM/IEEE Design Automation Conference (DAC), Austin, TX, June 2016.
Author's addresses: Y. Song and B. Lin, Electrical and Computer Engineering Department, University of California San Diego; K. Samadi, Qualcomm Research.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© YYYY ACM. 1084-4309/YYYY/01-ARTA $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

performance. In particular, competing memory requests from different cores interfere with each other, and these memory interferences can lead to a slowdown of application performance due to memory stalls or a failure to sustain real-time performance due to inadequate memory bandwidth. Therefore, an effective memory controller that can carefully reconcile memory interferences is of utmost importance because it plays a central role in meeting the QoS needs of real-time cores while maximizing the performance of applications running on CPUs.

A typical real-time core such as a GPU consumes much higher bandwidth than a CPU, because a GPU is capable of executing multiple threads in parallel, which often leads to a large number of memory requests. On the other hand, a real-time core like a GPU can switch between different threads to hide memory access latency. In contrast, the sparse CPU traffic after the last-level cache is typically much more sensitive to memory stalls. Memory interference happens when the bandwidth-intensive real-time traffic overwhelms the latency-sensitive CPU traffic. Previous works have analyzed the heterogeneous memory traffic in details [Power et al. 2013; Ausavarungnirun et al. 2012; Jeong et al. 2012; Lee et al. 2013; Mishra et al. 2013].

A classic memory controller for multi-core platforms comprises a two-tier queuing system, including a per-source queuing transaction stage and a per-bank queuing command stage. The two-tier queuing system is a straightforward solution to deliver high memory throughput. To accommodate heterogeneous traffic, Ausavarungnirun et al.

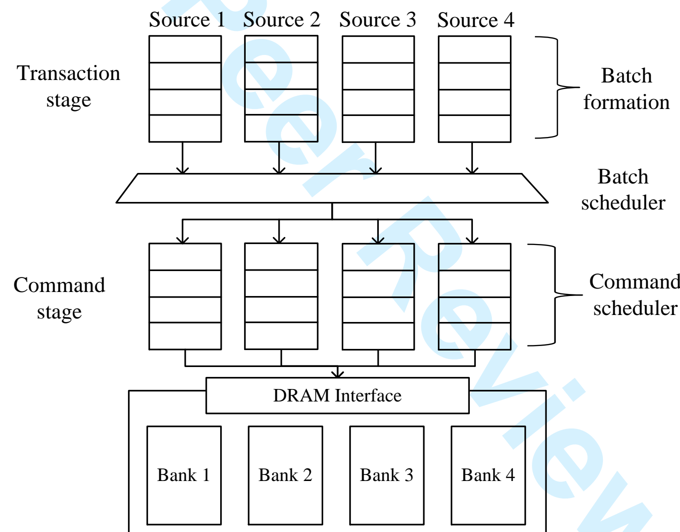


Fig. 1. The architecture of a two-tier staged memory scheduler.

[Ausavarungnirun et al. 2012] were the first to propose the staged memory scheduler (SMS) in the context of integrated CPU-GPU systems (Fig. 1). The proposed SMS architecture also follows the two-tier queuing system approach. In SMS, transactions for the same row-buffer in DRAM are formed into batches to increase row-buffer hits. Due to the lower spatial locality, CPU traffic generates shorter batches. By adopting the Shortest Job First (SJF) policy with certain probability, the scheduler prevents CPU traffic from being overwhelmed by the vast GPU traffic. However, the command stage still uses conventional FIFOs, which degrades the QoS improvement from the transaction stage. In particular, once scheduled transactions have been translated into com-

mands and forwarded to the second stage, newly arriving memory transactions that may be more critical from a QoS perspective cannot be serviced ahead of those translated commands that are already queued at the second stage. As a concurrent work, Jeong et al. [Jeong et al. 2012] proposed a real-time QoS-aware scheduling algorithm for the transactions stage, but it also neglects the queuing effects at the command stage.

In both of these prior works [Ausavarungnirun et al. 2012; Jeong et al. 2012], the proposed memory scheduling policies aim to minimize the CPU traffic latency while ensuring the required bandwidths of the real-time cores are satisfied. Effectively, memory scheduling can be depicted as the optimization problem in (1), where L_{cpu} refers to the latency to CPU traffic and $R_{real-time}$ represents the required bandwidths of real-time cores, and FPS (Frame Per Second) is the target frame rate. The CPU can be replaced by any other non-real-time cores and the real-time cores may include the graphic GPU, the HD video coder, and the display since all of them may share the same off-chip memory. A good memory controller minimizes the queuing delay of CPU traffic while providing sufficient memory bandwidth to real-time cores.

$$\begin{array}{rl} \min & L_{cpu} \\ \text{subject to} & R_{realtime} \geq FPS * framesize. \end{array} \tag{1}$$

The drawback of previous works using two-tier queuing systems lies in the unawareness of queuing delays in the command stage which can be much higher than that in the transaction stage. Performance improvement by the transaction scheduler may not be preserved after the command stage. This problem with two-tier queuing is illustrated in Fig. 2. The figure depicts a common scenario in which the GPU transaction queue has transactions to send but the CPU has no transactions available. As there is no CPU transaction waiting, a QoS-aware scheduler in a two-tier queuing system would send all the GPU transactions queued at the transaction stage to the command stage in order to maximize memory throughput. Yet any new incoming CPU transaction going to the same queue would have to wait at the end of the same command queue, which could cause potentially substantial latency to the CPU memory traffic.

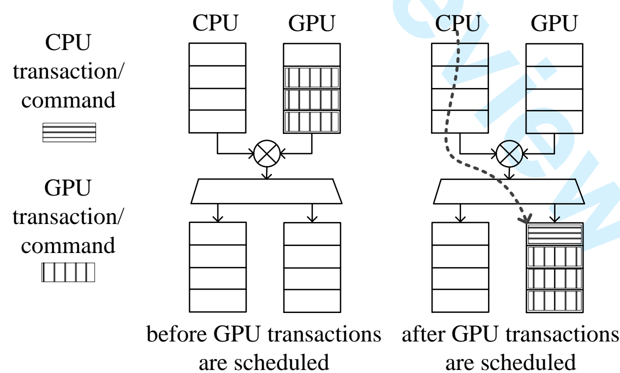


Fig. 2. An example of transaction scheduling in a two-tier memory controller leading to queuing delay to the CPU.

The problem is further illustrated in Fig. 3 where queuing delays in the transaction and command stages of a two-tier system are shown respectively. For comparison, three scenarios are simulated: one only has a dual-core CPU; another has a dual-core

For Peer Review

CPU, a graphic GPU, and a display controller, and uses the SJF policy for scheduling; the last one also has a dual-core CPU, a graphic GPU, and a display controller, but uses the Round-Robin (RR) policy for scheduling. Even though the SJF policy successfully reduces the queuing delay of CPU transactions, the queuing delay of CPU commands remains the same with the result from RR policy, which is much higher than the command delay without real-time cores. In other words the benefit introduced by the SJF policy in the first stage is counteracted by the FIFO queues in the second stage. Therefore, we say that a two-tier queuing system is *not efficacious* in the sense that a QoS-aware scheduler may not produce the desired result.

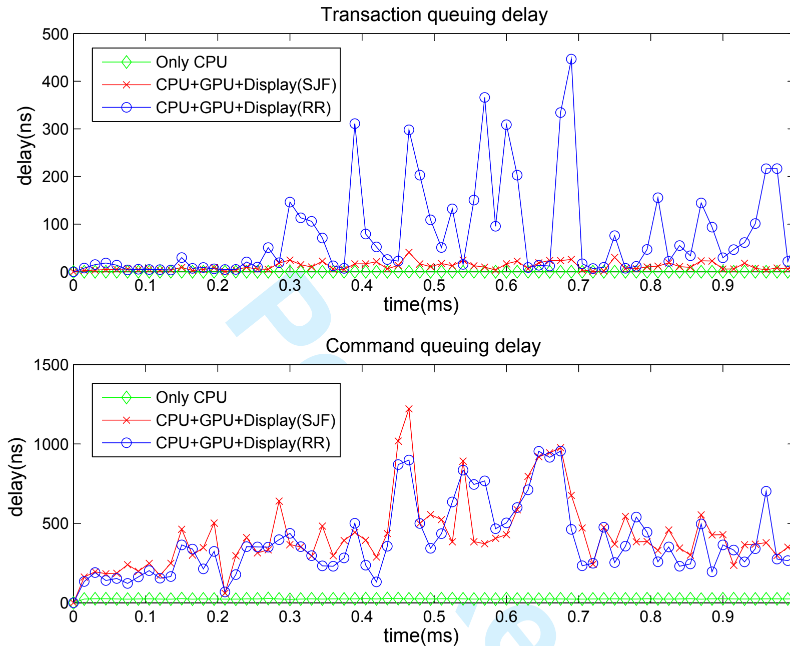


Fig. 3. Queuing delays of CPU traffic in the transaction and command stages of a two-tier memory controller. *mcf-art* and *coc* benchmarks used for CPU and GPU. Experiment setup discussed in Section 6.

In this work, we propose a new memory controller architecture with single-tier virtual queuing for multiple heterogeneous cores. The aim of the STVQ controller is to optimize system performance in (1). The contributions of our work can be summarized as follows.

— **Limitation of Two-Tier Queuing Systems for MPSoCs**: We identify the bottleneck of two-tier memory schedulers in delivering good system performance for heterogeneous systems where the CPU and real-time cores have different requirements on latency and bandwidth.
— **Single-Tier Virtual Queuing (STVQ) memory system**: To get rid of command queuing delays and apply efficacious scheduling policies, we propose the single-tier virtual queuing memory controller as the solution for heterogeneous memory scheduling.
— **Separable allocation for scalable arbitration**: To achieve high scalability, separable allocation is used to implement the selection of memory bank and virtual queue to service, which reduces the complexity of arbitration.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

— **Multi-source real-time scheduling**: To modulate traffic flows from the CPU and real-time cores, including the GPU, the HD video coder, and the display, we present a QoS-aware scheduling policy for heterogeneous memory traffic from multiple real-time cores.
— **Row-buffer hit optimizations**: To increase row-buffer hits and reduce average memory latency, we further extend the scheduling policy to support a batching mechanism.

Based on our experimental results with the proposed real-time scheduling algorithm, we show that the proposed STVQ memory controller successfully reduces slowdown of the CPU caused by the memory interference from real-time cores. Compared with previously proposed QoS memory controllers, our approach reduces CPU slowdown by up to 13.9%, with an average reduction of 6.1%, while ensuring that all real-time cores are able to meet their frame rate targets. With batching applied in the STVQ controller, we further show that row-buffer hits can be increased by up to 66.2% and on average by 43%. As the result of these row-hit improvements, the memory latency for the overall traffic can be reduced by up to 19.8%, with an average reduction of 9.9%, while only affecting the CPU by a negligible 0.9% degradation in IPC.

The rest of this paper is organized as follows: Section 2 briefly reviews related works. Section 3 describes our proposed single-tier virtual queuing memory controller architecture. Section 4 presents our multi-source real-time scheduling algorithm for the STVQ memory controller. Section 5 discusses the performance of the STVQ controller on DRAM efficiency and introduces batching as an extension of our scheduling policy. The experimental results and conclusions follow in Sections 6 and 7, respectively.

2. RELATED WORK

The First-Ready First Come First Serve (FR-FCFS) [Rixner et al. 2000] has been widely used for memory scheduling to deliver high memory throughput, but it assigns more priorities to memory-intensive applications as it prioritizes requests that result in high row-buffer hits. Application-aware scheduling for CPU-only systems has drawn more attention in recent years [Kim et al. 2010a,b; Jalle et al. 2014]. ATLAS [Kim et al. 2010a] prioritizes applications with low memory intensity to improve system throughput at the expense of applications with high memory intensity. Thread cluster memory scheduling [Kim et al. 2010b] dynamically clusters applications into low and high memory-intensity clusters and achieves high system performance and fairness simultaneously. Dual-criticality memory controller [Jalle et al. 2014] handles memory contention between real-time and high performance applications by bank separation, assuming no shared memory between different types of applications. Minimalist Open-page policy [Kaseridis et al. 2011] adopts a fair DRAM address mapping scheme to avoid row-buffer locality starvation and to increase bank level parallelism. On basis of that, the scheduler prioritizes requests with a higher criticality, and then it prioritizes requests that are hitting an open page. Similarly, parallelism-aware batch scheduling [Mutlu and Moscibroda 2008] batches requests in time order before prioritizing row-hit requests within each batch to circumvent starvation.

Staged memory scheduler [Ausavarungnirun et al. 2012] was the first application-aware scheduler proposed for CPU-GPU systems. A two-tier staged memory architecture (Fig. 1) is used to decouple the memory scheduling task into three basic steps, including batch formation, batch scheduling and command scheduler. The batch scheduler switches between two policies which are the Shortest Job First (SJF) and Round-Robin (RR). Due to the distinct characteristics of CPU and GPU memory traffic, SJF policy gives higher priority to the CPU while RR policy favors the GPU. By adjusting the probability of SJF policy, the priority of CPU is modulated to balance system per-

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

formance and fairness for the CPU-GPU system. However, even though the CPU can achieve sufficient bandwidth through the batch scheduler, there is no guarantee for the latency because the queuing delay in the FIFO command scheduler is uncertain.

In [Jeong et al. 2012], a QoS-aware scheduling policy was proposed to dynamically balance CPU and GPU bandwidth. The proposed policy allocates the minimum bandwidth to the graphic GPU to meet the target FPS. To attain that, two notions for GPU's frame progress were introduced by (2),

$$P_A = \frac{number\ of\ tiles\ rendered}{total\ number\ of\ tiles}, \quad P_E = \frac{elapsed\ time\ in\ current\ frame}{target\ frame\ time}. \tag{2}$$

Here, P_A is the actual progress of GPU workload in the current frame and P_E is the expected progress. Their QoS policy prioritizes the GPU when P_A is lagging behind P_E. Progress notions can be extended to include other real-time cores such as an HD decoder whose progress metric can be expressed as follows:

$$P_A = \frac{number\ of\ bytes\ (or\ pixels)\ processed}{total\ number\ of\ bytes\ (or\ pixels)}. \tag{3}$$

Nonetheless, this QoS policy focuses on transaction-level scheduling and optimizes CPU bandwidth instead of CPU latency, a more critical performance metric. This may cause performance degradation in a memory scheduler with the two-tier queuing system.

3. SINGLE-TIER VIRTUAL QUEUING MEMORY SYSTEM

A memory system in a single memory channel consists of the physical memory (DRAM devices) and a memory controller. As the bridge between processors and memory, the memory controller deals with demands from processors (bandwidth and latency etc.) while following DRAM memory-access protocols. The objective of our STVQ memory controller is to minimize latency of CPU memory traffic while satisfying frame rate requirements of real-time traffic. Fig. 4 shows an STVQ controller for a DRAM with N memory banks and K independent traffic flows. Four major components reside in the STVQ controller, including sing-tier virtual queues (VQ), bank arbiter, transaction scheduler and per-bank command generators. In this section, we will go through design details of the proposed single-tier virtual queuing memory system, followed by a brief overview of DRAM memory-access protocols in preparation for our scheduling algorithms in the next section.

3.1. Interface for Incoming Traffic Flows

At the input of the memory controller, incoming memory traffic flows from different sources are separated so that QoS-aware scheduling policies can be applied by the transaction scheduler. A real-time core can generate more than one independent traffic flows. In a tile-based graphic GPU, a complete frame is divided into a few tiles which can be rendered in parallel. A typical rendering procedure goes through the geometry shader, rasterizer and fragment shader etc. until the tile cache is flushed into the framebuffer in the external memory. The memory traffic generated during the rendering procedure of current tile, which is mostly input traffic of texture, does not depend on the flushing traffic of previous tiles. Thus GPU memory traffic can be split into rendering and flushing traffic flows. Similarly, an HD video decoder generates reading and writing traffic flows to meet the target frame rate respectively. The generation of real-time traffic can be shown as ideal pipelines in Fig. 5 where the delays

For Peer Review

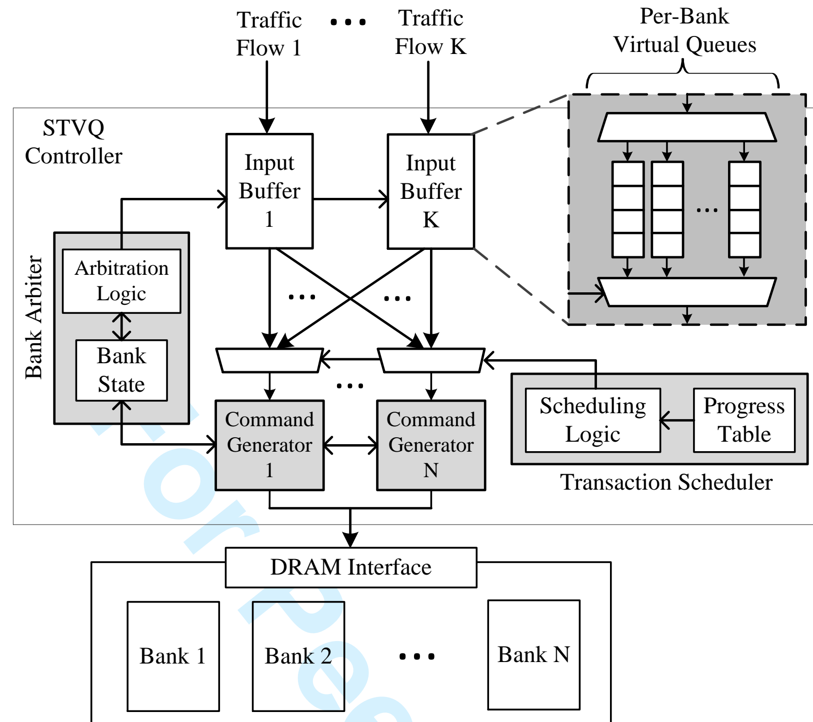


Fig. 4. The design of single-tier virtual queuing (STVQ) memory controller.

between consecutive stages are omitted. Each traffic flow is scheduled in parallel in the STVQ controller. In our experiment, real-time traffic flows are independent from each

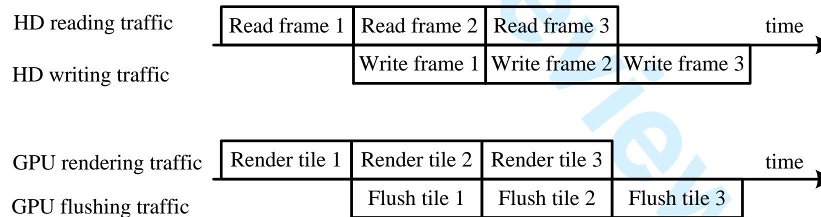


Fig. 5. The real-time input and output traffic of GPU and HD decoder.

other during memory scheduling while traffic from different CPU cores are combined together. Although it would allow to apply different scheduling policies to CPU cores by separating their traffic flows, we currently do not do that because the focus of our memory scheduling is to reconcile the conflicts between CPU and real-time memory traffic.

3.2. Single-Tier Virtual Queues

In the first stage of the STVQ memory controller, transactions from the same traffic flow are stored in the same input buffer. An input buffer contains a series of per-bank

virtual queues. Upon arrival, transactions are sorted into virtual queues based on the destination bank. By using per-bank queues, the STVQ controller is able to schedule transactions to different banks separately.

Since in every clock cycle, the memory controller can only send one command to the DRAM and receive one transaction from each traffic flow, multiple reading or writing does not happen to an input buffer. This allows per-bank virtual queues for the same traffic flow to share one buffer without raising storage cost. In the implementation, virtual queues can be statically allocated, which requires a simple address table to record the start address of each virtual queue inside the input buffers. In a conventional memory controller, the two-tier queuing system guides memory traffic from per-source buffers to per-bank buffers. In spite of its high scalability, it takes extra effort to unify the scheduling and queuing policies in two stages. In our case, by consolidating two-tier queues into single-tier queues, scheduling is more efficient and fewer buffers are used.

3.3. Separable Allocation

To schedule a transaction to DRAM, the STVQ controller selects among per-flow per-bank queues, which is an allocation problem where access to the DRAM interface is granted to one of the virtual queues. Similar to *separable allocators* in on-chip network routers [Becker and Dally 2009], we achieve high scalability by performing allocation with two separable steps: one to choose a memory bank and the other to choose a traffic flow.

In the first step, the bank arbiter assigns DRAM access to one of the memory banks. Meanwhile, virtual queues for this bank are activated in all input buffers and contend for the designated command generator which will convert transactions into DRAM commands. Note that the command generator may receive requests from multiple traffic flows. Therefore, in the second step, the transaction scheduler arbitrates among available traffic flows at the command generator. Only then can the chosen virtual queue send a transaction to the command generator.

By using separable allocation, the STVQ controller solves a simple N:1 or K:1 arbitration problem at a time, where N is the number of banks and K is the number of traffic flows. For example, if there are 4 traffic flows and 8 memory banks, that requires the STVQ controller to perform an 8:1 arbitration to choose a bank, and then a 4:1 arbitration to select a traffic flow. The use of separable allocation significantly reduces the complexity of the arbitration problem.

3.4. Bank Arbiter

In a DRAM device, memory banks operate separately at the same time. For example, bank 1 can be read while bank 2 is being written. Yet since they share the same interface to the memory controller, only one memory bank can receive DRAM command in the same clock cycle. Also, the memory bank is not available for more commands until the current operation has finished. Therefore, the bank arbiter is responsible to find available memory banks and assign one of them with the permission to access DRAM interface.

To achieve this, the arbiter records bank states in a state table and tracks the availability of each memory bank. The arbitration logic selects a winner from available banks based on a certain arbitration policy. In our experiment, the bank arbitration policy does not have notable impact on the performance. Hence we adopt the round-robin policy for simplicity. After arbitration, the bank arbiter sends the bank selection signal to input buffers which activates the corresponding virtual queue for the winning bank and forwards the request to the command generator. Meanwhile the command generator which is designated to the winning bank is also activated.

For Peer Review

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60

3.5. Transaction Scheduler

The transaction scheduler arbitrates among requests from different traffic flows at the command generator after bank arbitration. The winning queue schedules a transaction to the command generator afterwards. The traffic flow arbitration and transaction scheduling are performed by the scheduling logic which implements our multi-source real-time scheduling algorithm (discussed in the next section). As input to the scheduler, progress information (2,3) of real-time traffic flows is monitored and recorded by the progress table.

3.6. Command Generators

After bank arbitration and transaction scheduling, only one transaction arrives at the command generator which is designated to the selected memory bank. To execute this memory transaction, the command generator generates commands to operate DRAM while following the DRAM memory-access protocol [Jacob et al. 2007].

The major task of a command generator is to deal with various timing constraints at the command level. Thanks to the command generators, the bank arbiter and transaction scheduler are able to perform allocation without dealing with the intricate DRAM access protocol. Also, per-bank command generators communicate with the bank state table in the bank arbiter to update the bank availability information. In STVQ a command generator is implemented as a state machine which is similar to the command schedulers in prior works [Ausavarungnirun et al. 2012; Rixner et al. 2000; Kim et al. 2010a,b; Jalle et al. 2014].

3.7. DRAM Access Protocol

A single read transaction includes three commands/operations: bank activation, column read and precharge. During command scheduling, consecutive commands have to be separated by minimum intervals in time so that they can in turn engage the different shared resources in the DRAM, such as I/O gating and sense amplifiers. For example, during a single read operation, t_{RCD} specifies the minimum time interval between row activation and column read, and t_{CL} is the interval between column read and data burst on the data bus. In a more common situation where multiple access operations are performed on an open row in a memory bank, t_{WTR} is the minimum interval between the end of a write command and the start of a read command.

Furthermore, due to dynamic power concerns, there are constraints on the frequency of bank activations in the same DRAM device. The constraint t_{FAW} determines a rolling time window during which no more than four banks can be activated. Another constraint t_{RRD} specifies the minimum interval between consecutive row activations in the DRAM device. Finally, refresh commands are needed every several thousand cycles to maintain the stored data in DRAM arrays. More illustrations of DRAM memory-access protocol can be found in [Jacob et al. 2007].

3.8. Hardware Implementation

3.8.1. Virtual Queues. As explained, virtual queues of the same traffic flow share the same input buffer. For an STVQ memory system with N memory banks and K traffic flows, only K buffers are needed. By comparison a two-tier memory system requires $N+K$ buffers because all the queues in Fig. 1 can be active at the same time. Moreover, command queues in the two-tier system can take up lots of storage, because every transaction may generate up to three DRAM commands. By avoiding buffering DRAM commands, more storage can be saved in the single-tier design. The storage of an input buffer can be equally allocated to per-bank virtual queues. To record the addresses of

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

N virtual queues in each buffer, N registers are needed. Note that all input buffers can share the same address table for virtual queues.

3.8.2. Bank Arbiter. The logic of bank arbitration is similar to previous memory schedulers. To implement round-robin policy, the arbiter maintains a register recording the last serviced memory bank. A small state table is used to track availability of memory banks. Each bank has a one-bit flag to indicate its availability.

3.8.3. Transaction Scheduler. Since the arbitration for transactions is performed for a single memory bank each time, one transaction scheduler is enough in our design. The progress table provides scheduler with progress information of real-time traffic. If there are K traffic flows, there should be at most $K-1$ real-time traffic flows with the other one from the CPU. Each real-time traffic flow has two types of fractional progress, i.e. P_A and P_E (2,3). For each type of progress, two registers are used to store the denominator and the numerator. Specifically, to compute P_E, a register records the target frame time and the other register keeps track of elapsed time in terms of clock cycles. To compute P_A for an HD decoder, a register records the total number of bytes to be processed in current frame and the other updates the number of bytes which have been processed in memory. The computation of P_A for a graphic GPU requires the tile engine in the GPU to output tiling information including the number of tiles rendered and the total number of tiles in current frame.

3.8.4. Command Generators. A command generator is implemented for each bank as a state machine performing scheduling based on various timing and power constraints, which is identical to previous schedulers [Ausavarungnirun et al. 2012; Rixner et al. 2000; Kim et al. 2010a,b; Jalle et al. 2014]. As suggested in [Zhang et al. 2014] we actually only need four command generators, because no more than four memory banks can be active at the same time due to the constraint t_{FAW}. Four command generators are shared by active memory banks. Yet to implement that more arbitration logic is required to allocate command generators to active memory banks. Thus in our experiment we use per-bank command generators.

3.8.5. Hardware Cost. The hardware parameters of the STVQ memory controller employed in our experiment are configured as follows. We use an STVQ controller working at 1GHz, scheduling memory traffic for eight memory banks, and receiving six traffic flows including two flows from the graphic GPU, two from the HD decoder, one from the display control and one from CPU cores. Each virtual queue reserves 8 entries in the input buffer. With 48 virtual queues in total, up to 384 transactions can reside in the controller, which is not expensive compared with two-tier design [Ausavarungnirun et al. 2012]. In the progress table, a real-time traffic flow has two entries including P_A and P_E, each of which is a fractional number with two integer components. We assume the lowest admissible frame rate is 23FPS (for movies), which means the maximum frame time contains no more than 64 millions clock cycles. Thus a 26-bit register is enough to record the frame time. Two of such registers are used to store a fractional real-time progress. The hardware storage overhead in the STVQ controller is summarized in Tab.I.

4. MULTI-SOURCE REAL-TIME SCHEDULING

In this section we will discuss our QoS-aware scheduling algorithm for multiple real-time traffic flows on basis of the proposed STVQ memory controller. As in (1), the scheduling objective for the STVQ controller is to minimize latency for CPU traffic while providing real-time traffic required bandwidth. To guarantee real-time traffic obtains sufficient bandwidth, the transaction scheduler monitors frame progresses (2,3) of each real-time traffic flow. A real-time virtual queue sends transactions to DRAM

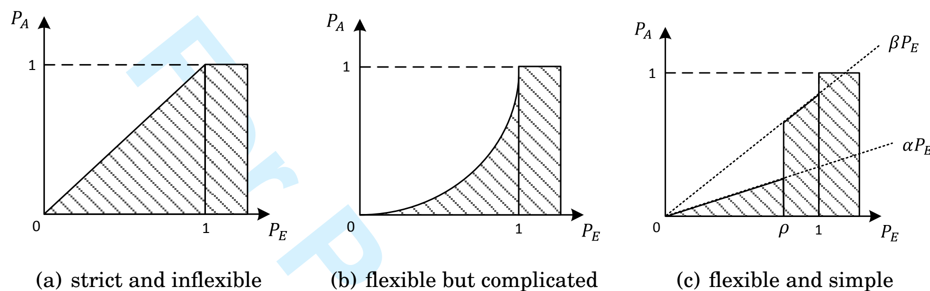
For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

Table I. Hardware storage in the STVQ controller with six traffic flows ($K = 6$) and eight DRAM banks ($N = 8$).

Storage	Description	Size
Per-bank VQ	Buffers transactions from a single flow going to a single bank	8 entries
Input buffer	Stores transactions from a single flow	$N \times VQ_Size = 64$ entries
VQ address table	Records each virtual queue's start address	$N \times log_2 Buffer_Size = 48$ bits
Bank state table	Tracks the availability of each memory bank	$N \times 1 = 8$ bits
Progress table	Monitors progresses of real-time traffic flows	$(K-1) \times 2 = 10$ entries
Progress entry	Represents a fractional progress	$2 \times log_2 FrameTime_{max} = 52$ bits

when the actual progress falling behind the expected progress. Meanwhile, the CPU receives service as often as possible to reduce queuing delay at the corresponding virtual queue.



(a) strict and inflexible (b) flexible but complicated (c) flexible and simple

Fig. 6. Priority zone of a single real-time traffic flow.

For Peer Review

The priority zone of real-time traffic is shown in Fig. 6 where the horizontal axis represents the expected progress (P_E) of a real-time traffic flow and the vertical axis is for the actual progress (P_A). The shaded zone indicates when the real-time traffic obtains priority over CPU traffic, namely, when the actual progress is deemed to be inadequate relative to the expected progress. The expected progress reaches 1 when current frame comes to the end. After that the real-time traffic obtains absolute priority. In a strict implementation as in Fig. 6(a), the actual progress (P_A) is forced to keep up with the expected progress (P_E) which grows linearly over time. This implementation does not have enough flexibility. When CPU traffic has transactions available, it may be more desirable to serve the CPU while postponing real-time traffic when the current frame will not finish any time soon. Thus in a flexible implementation shown in Fig. 6(b), real-time traffic has less priority than Fig. 6(a) in the beginning so that the CPU has more opportunities to be served. Note that this will not cause starvation to the real-time traffic because CPU traffic cannot constantly interrupt real-time traffic even with higher priority, thanks to the sparse traffic pattern. In our experiment, we find it unnecessary to use high order priority functions to achieve good performance. Instead a linear piecewise function $f(P_E)$ (4) is implemented. As shown in Fig. 6(c) real-time traffic obtains priority when $P_A < f(P_E)$. The tunable parameters are empirically determined as $\alpha = 0.5, \beta = 0.95, \rho = 0.95$.

$$f(P_E) = \begin{cases} \alpha P_E & P_E \in [0, \rho) \\ \beta P_E & P_E \in [\rho, 1) \\ 1 & P_E \in [1, \infty) \end{cases} \quad (4)$$

The scheduling algorithm used for multiple real-time traffic flows is shown in Algorithm 1. The real-time traffic achieves the absolute priority when it is supposed to

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

ALGORITHM 1: Transaction Scheduling for the CPU and Multiple Real-time Traffic Flows

Input: P_E^i, P_A^i, $ratio^i = \frac{P_A^i}{P_E^i}$ for each real-time queue $i = 1, ..., n$.

1: find the queue u with the lowest ratio, i.e. $ratio^u = min\{ratio^i | i = 1, ..., n\}$
2: find the queue v with the highest expected progress, i.e. $P_E^v = max\{P_E^i | i = 1, ..., n\}$
3: **if** $ratio^u \geq 1$ **then**
4: serve CPU transaction queue.
5: **else if** $P_E^v > \rho$ **and** $ratio^v < \beta$ **then**
6: serve real-time transaction queue v.
7: **else if** CPU transaction available **and** $ratio^u > \alpha$ **then**
8: serve CPU transaction queue.
9: **else**
10: serve CPU and real-time queue u in round-robin manner.
11: **end if**

For Peer Review

finish the current frame soon ($P_E > \rho$), but its actual progress has not caught up yet ($P_A < \beta P_E$) (Line 6). The CPU obtains priority when it has requests available while the real-time traffic is not too far behind its expected progress, i.e. $P_A > \alpha P_E$ (Line 8), or P_A is leading P_E for all real-time traffic (Line 4). Otherwise, the real-time and CPU traffic share equal priority (Line 10). Note that this algorithm allows real-time traffic to be late for a few cycles which is not noticeable for a graphic application whose frame period is on millisecond level.

5. MEMORY PERFORMANCE OPTIMIZATION

The heterogeneous memory scheduling problem in (1) is focused on the performance of CPU and real-time cores, but in practice the performance and efficiency of the DRAM memory system can also be a concern. Since a mobile device has limited battery life, the DRAM memory, as a major power consumer, needs to provide as much data transfer as possible within certain time and power limits. In this section, we will discuss DRAM efficiency and further explore batching techniques in the STVQ memory controller.

5.1. DRAM Efficiency

DRAM efficiency [Jacob et al. 2007] is defined as

$$\text{DRAM efficiency} = \frac{\text{data transfer time}}{\text{active time}} \quad (5)$$

where the data transfer time refers to the cycles when a DRAM device transfers data on the bus, and the active time is the total runtime of the DRAM device which can be either transferring data or processing DRAM commands.

DRAM efficiency is usually lower than 100% because it takes a DRAM memory bank extra cycles to prepare for data transfer, such as activating row-buffer (t_{RCD}) and precharging (t_{RP}). Yet, theoretically 100% efficiency is achievable through perfect address mapping to enable high bank-level parallelism, which hides the latency of data transfer preparation. This way, there is always a memory bank ready for data transfer while others being occupied. However, address mapping is beyond the context of memory controller design.

A memory controller can achieve a high DRAM efficiency by reconciling memory interference among traffic flows so that more data is transferred in the same amount of time. Compared with other schedulers, the STVQ memory controller helps improve DRAM efficiency by allowing more CPU transactions to be processed, while providing the required bandwidth to real-time cores.

5.2. Row-Buffer Hits and Batching

In [Ausavarungnirun et al. 2012], consecutive entries in the same transaction queue and to the same row-buffer are batched together and scheduled to DRAM memory in succession. These transactions read or write columns in the same row-buffer once the row is activated. By allowing multiple row-buffer hits after one activation and saving the time which would otherwise be used to precharge and re-activate the same row, batching reduces the average memory access latency and also the dynamic power.

With higher spatial locality, real-time traffic flows can form into larger batches than CPU traffic. Thus real-time cores gain more benefits from batching than the CPU. The downside of batch formation is that when a real-time transaction batch is being scheduled, it cannot be interrupted even if a CPU transaction with a higher priority becomes available. That will cause extra queuing delay to CPU traffic in the transaction queue and thus deteriorate the CPU IPC.

In the STVQ controller, open page policy is adopted for the row-buffer management. A row-buffer is activated until the next transaction is scheduled to a different row in the same bank. This way consecutive transactions to the same row naturally result in multiple row-buffer hits even without explicit batch formation. A potential concern is that, without batch formation a real-time traffic may often be interrupted by the traffic with a higher priority and thus suffer from low row-buffer hits. This, however, does not happen in practice. On one hand, sparse CPU traffic can hardly cause interference to real-time traffic. On the other hand, the priority interchange between real-time traffic flows should not be frequent, as a result of the limited granularity of progress notions. For example, a graphic GPU updates the progress notions in (2) each time after a tile has finished processing (about $1us$). Only after that can the GPU traffic flows update their priorities. Since a column access operation takes around $10ns$, multiple row-buffer hits can easily happen between the priority updates.

Moreover, the STVQ controller can still be equipped with batching mechanism to further increase row-buffer hits if the degradation on CPU memory latency can be accepted to some extent. The scheduling policy in Algorithm 1 can be simply modified to include batch formation (shown in Algorithm 2). To be specific, we need a register for each memory bank to record the index of the active row-buffer and a counter to record the number of transactions in current batch. When a transaction is picked up by the transaction scheduler, the batch formation is performed to find other transactions in the same batch by comparing the destination row index. Up to N_{max} transactions can be included in a single batch. After batching, the number of transactions in current batch is stored and decremented every time when a transaction is issued from the batch. Transaction scheduling is performed again only after current batch has been finished.

Note that for heterogeneous memory scheduling, batching is not meant to improve DRAM efficiency. In a heterogeneous system, most memory traffic comes from real-time cores whose average memory latency can be reduced greatly by batching. However, the lower memory latency for real-time traffic does not lead to more transactions to be scheduled. This is because the number of real-time transactions scheduled in each frame period depends on the frame size. Lower memory latency means the frame can be finished sooner, but a real-time core still has to wait for the frame period to end before starting a new frame. As a result, the DRAM efficiency (5) remains the same for real-time traffic.

6. EVALUATION

The primary responsibility of a heterogeneous scheduler is two-fold: allocating required bandwidth to real-time cores and keeping the slowdown of the CPU as low as

ALGORITHM 2: Transaction Scheduling with Batch Formation

Input: the transaction queue currently being served, q_{cur}; the number of transactions left in current batch, N_{cur}; the maximum number of transactions allowed in a batch, N_{max}
1: **if** $N_{cur} == 0$ **then**
2: use Algorithm 1 to find a new queue, q_{new}
3: assign $N_{cur}(\leq N_{max})$ with the number of neighbor transactions in q_{new} going to the same row-buffer
4: serve q_{new}
5: $q_{cur} = q_{new}, N_{cur} --$
6: **else**
7: serve q_{cur}
8: $N_{cur} --$
9: **end if**

Table II. Simulation Parameters.

CPU Parameter	Description
Clock speed	1GHz
L1 cache	32KB private 4-way
L2 cache	1MB shared 16-way
GPU Parameter	Description
Unified shader cores	8
Clock speed	800MHz
DRAM Parameter	Description
Volume	4GB
I/O bus clock	666.67MHz
CL-tRCD-tRP	10-10-10
tWTR-tRTP-tWR	5-5-10
tRRD-tFAW	4-20
Channels-Ranks-Banks	1-1-8

possible. In this section, the proposed STVQ memory controller will be first tested on bandwidth allocation, CPU slowdown reduction and DRAM efficiency without batching techniques, in comparison with previous schedulers for heterogeneous memory traffic. Further we will explore batching mechanism in the STVQ controller. The power and area analysis of the STVQ memory controller follows in the end.

6.1. Methodology

Our simulation methodology for the STVQ memory controller is based on DRAMSim2 [Wang et al. 2005], on top of which a heterogeneous system is simulated by a trace-based simulator, including a dual-core CPU, a graphic GPU, an HD decoder, and a display controller. The DRAM model is configured according to the DDR3 SDRAM-1333 specification. The simulation parameters are listed in Table II. The realtime cores can either include the GPU and display controller, or the HD decoder and display controller, since on a mobile device the display is usually engaged by either the GPU or HD decoder.

For the test traces, we have four CPU applications from SPEC 2000/2006 benchmarks with different memory intensities, including *art*, *mcf*, *hmmer* and *sphinx3*. The GPU traces are extracted from the instrumented softpipe driver in the open-source OpenGL library Gallium3D [Graphics 2010; Arnau et al. 2013b,a]. Four representative mobile games are selected from different genres. The memory trace for HD decoder is generated from a commercial MP4 parser. More benchmark descriptions can be found in Table III.

For Peer Review

Table III. Benchmark Descriptions.

GPU benchmark	Description	FPS
coc	strategy game	50
fruit	action game	60
minecraft	sandbox game	60
transformer	shooting game	60
Video benchmark	Description	FPS
interstellar, knight	4k ultra HD movie trailer	24
CPU benchmark	Description	Bandwidth
art	image recognition	1.06 GB/s
mcf	combinatorial optimization	0.94 GB/s
hmmer	biosequence analysis	0.18 GB/s
sphinx3	speech to text program	0.005 GB/s

6.2. Comparison of Memory Schedulers without Batching

6.2.1. Bandwidth Allocation. To begin with, we test our memory controller with the CPU, GPU and display control. For the dual-core CPU we use *mcf* and *art* benchmarks and *coc* for the GPU. The bandwidth allocation for the CPU and GPU during one frame period (20 ms) by the STVQ memory controller with the proposed scheduling policy is shown in Fig. 7 (blue curve with circular markers). For comparison two other schemes are implemented: one only has CPU cores generating memory requests (green curves with diamond markers); the other applies SJF scheduling (in favor of CPU) in a two-tier memory controller (red curves with cross markers).

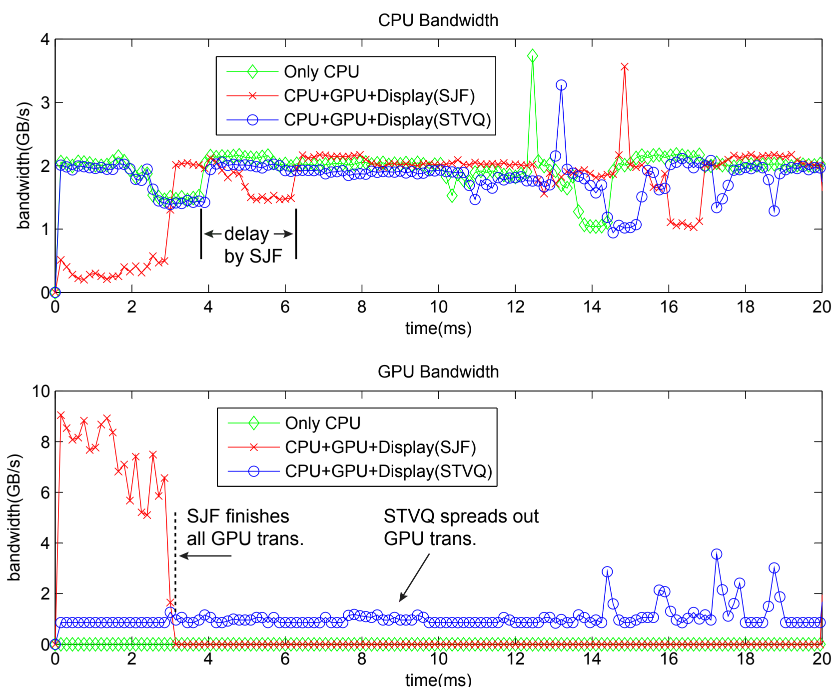


Fig. 7. Bandwidth allocation for CPU and GPU.

With no concerns for command queuing delay, the two-tier memory controller with SJF scheduling allows the GPU to quickly finish all the transactions in current frame time and consume most of the bandwidth in the first 3ms. At the same time applications on CPU cores hardly make any progress, leading to a delay lasting almost 3 ms which can be clearly observed between the red and green curves in the CPU bandwidth diagram. On the other hand, the STVQ memory controller keeps GPU bandwidth low and steady most of the time to alleviate memory contention against the CPU. In the GPU bandwidth diagram, spikes appear near the end of the frame period when the actual progress of GPU traffic is behind the expected progress and the GPU receives more priority in order to finish current frame in time. The CPU bandwidth diagram shows that the results of the CPU-only system and heterogeneous system with STVQ stay close to each otherbecause the STVQ memory controller does not cause much latency to CPU traffic. For clarity, the bandwidth of the display is not shown in Fig. 7 since it is very similar to the GPU's.

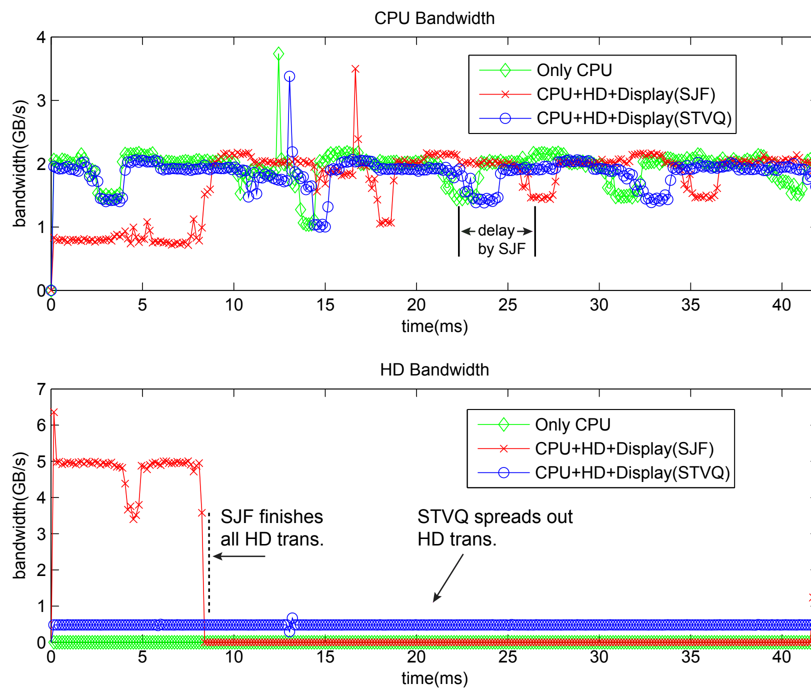
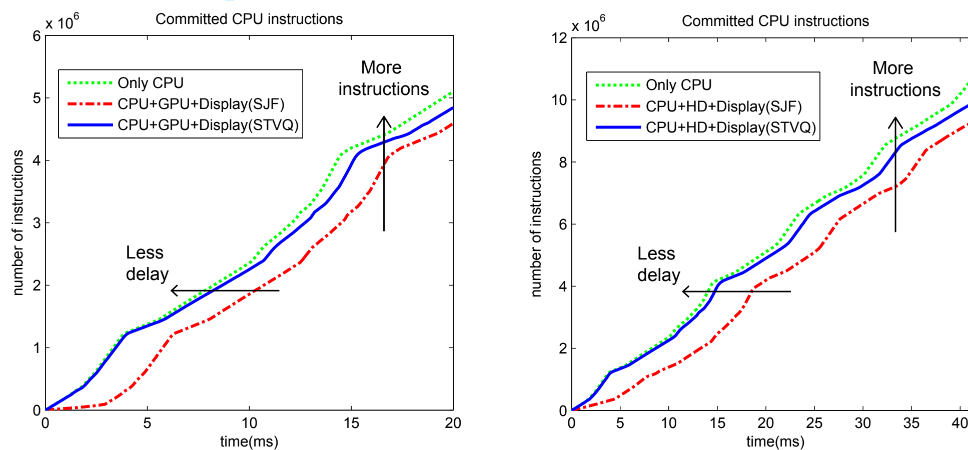


Fig. 8. Bandwidth allocation for CPU and HD decoder.

Further we test the STVQ memory controller with the CPU, the HD decoder and the display control. Benchmarks *mcf* and *art* are applied on CPU cores again, while the 4k HD movie clip *knight* is used to generate memory traces for the HD decoder. The bandwidth allocation by the STVQ controller during one frame period (42 ms) is shown in Fig. 8. Similar to the previous example, the two-tier scheduler using SJF policy gives more bandwidth to realtime cores in the beginning. As 4k movie has more pixels in each frame than GPU applications (1280×720 pixels), it takes the DRAM

more time to finish serving the HD decoder in Fig. 8 compared with Fig. 7. The delay caused to CPU applications can be observed by comparing CPU bandwidth with that of the CPU-only scenario. The STVQ controller leads to a much smaller delay for CPU traffic. Moreover, the STVQ controller evenly distributes bandwidth consumption of the HD decoder over the whole frame. Note that the bandwidth curve of the HD decoder in Fig. 8 is much smoother than that of GPU in Fig. 7. This is because the fine-grained pixel-based progress notion of HD decoder (3) facilitates more accurate realtime scheduling than the tile-based GPU progress (2).

6.2.2. IPC Slowdown. Due to memory conflicts, the CPU endures lower IPC when sharing memory with realtime cores, namely, less CPU instructions are executed during the same amount of time. A good memory scheduler reduces IPC slowdown by decreasing memory latency for the CPU. To test the STVQ controller on IPC slowdown reduction, same comparison scenarios are used again including the CPU-only system and the heterogeneous system with a two-tier scheduler using SJF policy.



(a) interference from GPU and display control (b) interference from HD decoder and display control

Fig. 9. Committed CPU instructions during one frame period with memory interference from realtime cores.

Fig. 9 shows the committed CPU instructions during one frame period. Two combinations of realtime cores are simulated: Fig. 9(a) shows the CPU under interference from the GPU and display control memory traffic; Fig. 9(b) shows the CPU affected by the HD decoder and display control. Same to the last section, benchmarks *mcf-art*, *coc* and *knight* are applied to the CPU, GPU and HD decoder. As expected, the CPU-only system executes the most instructions over the same period compared with other scenarios. Meanwhile the STVQ memory controller commits more instructions than the two-tier memory controller. Note that the horizontal distance between two curves shows the time difference after finishing the same number of instructions, which indicates the delay of CPU traffic caused by memory interference. As can be seen, the STVQ memory controller results in less delay than the two-tier scheduler. In both cases, the two-tier scheduler leads to huge delays to the CPU in the beginning. In contrast, the STVQ controller has the CPU traffic delay increase gradually over time by alleviating more memory conflicts.

As an important metric of system performance, the slowdown rate is calculated as the ratio of the baseline IPC, when CPU has exclusive access to memory, to the IPC

when CPU shares memory with realtime cores. Lower slowdown rate is more desirable as the memory controller targets at minimizing memory latency for CPU. Besides the proposed STVQ memory controller, we have three other scheduling policies on the two-tier memory scheduler as in Fig. 1, including RR policy, SJF policy and an extended version of the QoS policy in [Jeong et al. 2012] to include multiple realtime cores which are served in round-robin manner when the CPU has low priority.

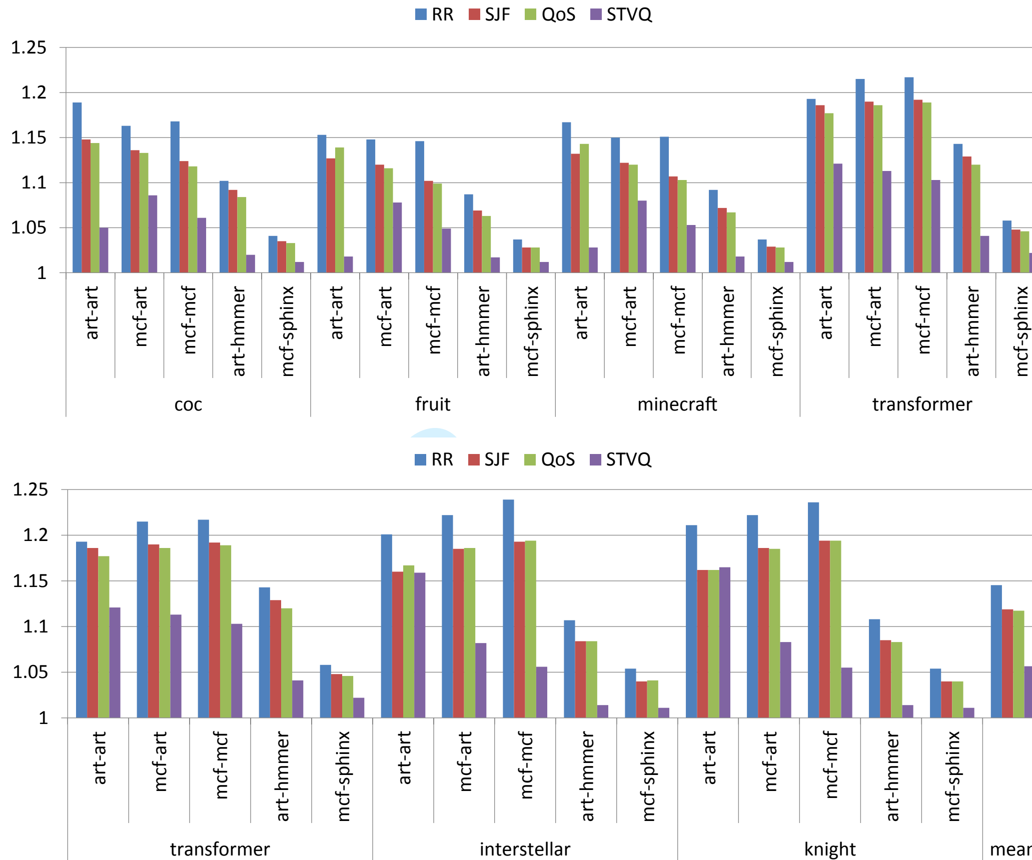


Fig. 10. IPC slowdown rate summary.

The summary of slowdown rates of all scenarios in 50 ms is shown in Fig. 10. As expected the RR has the worst performance because realtime traffic has far more transactions than the CPU and RR policy prefers the bandwidth-intense realtime traffic. Meanwhile SJF and QoS have similar results, because the latency of CPU traffic reduced in the transaction stage is counteracted by the much bigger queuing delay in the command stage. The STVQ memory controller achieves the best performance except for *art-art* and *knight*. In that case the slowdown rate by STVQ is a bit higher than SJF and QoS. This happens when adverse address mapping leads to plenty of bank conflicts between realtime and CPU traffic. Instead of letting bank conflicts happen all at once, the STVQ memory controller spreads out the conflicts over time, which in total may cause more latency to CPU traffic than the spared queuing delay. However, adverse

address mapping takes place rarely in our experiment even with random address allocation. With careful mapping between virtual, physical and DRAM addresses, severe bank conflicts can be avoided. Compared with the best performances of other schedulers, the STVQ memory controller lowers the slowdown rate by 6.1% on average. The highest reduction of slowdown rate by STVQ is 13.9% (*mcf-mcf* and *knight*).

Table IV. Performance summary of STVQ memory controller.

Realtime benchmark	CPU benchmark	Prev. IPC slowdown	STVQ IPC slowdown	Frame time slack (μs)
coc	art-art	114.4%	105.0%	0.0
	mcf-art	113.3%	108.6%	0.2
	mcf-mcf	111.8%	106.1%	0.2
	art-hmmer	108.4%	102.0%	0.0
	mcf-sphinx	103.3%	101.2%	0.2
fruit	art-art	112.7%	101.8%	0.6
	mcf-art	111.6%	107.8%	0.3
	mcf-mcf	109.9%	104.9%	0.2
	art-hmmer	106.3%	101.7%	0.4
	mcf-sphinx	102.8%	101.2%	0.3
minecraft	art-art	113.2%	102.8%	0.1
	mcf-art	112.0%	108.0%	0.6
	mcf-mcf	110.3%	105.3%	0.4
	art-hmmer	106.7%	101.8%	0.0
	mcf-sphinx	102.8%	101.2%	0.2
transformer	art-art	117.7%	112.1%	0.3
	mcf-art	118.6%	111.3%	0.1
	mcf-mcf	118.9%	110.3%	0.1
	art-hmmer	112.0%	104.1%	0.0
	mcf-sphinx	104.6%	102.2%	0.1
interstellar	art-art	116.0%	115.9%	18.1
	mcf-art	118.5%	108.2%	12.0
	mcf-mcf	119.3%	105.6%	18.1
	art-hmmer	108.4%	101.4%	5.9
	mcf-sphinx	104.0%	101.1%	5.9
knight	art-art	116.2%	116.5%	6.6
	mcf-art	118.5%	108.3%	13.4
	mcf-mcf	119.4%	105.5%	26.8
	art-hmmer	108.3%	101.4%	6.5
	mcf-sphinx	104.0%	101.1%	6.5

Frame time slacks ($:= \frac{1}{FPS_{target}} - \frac{1}{FPS_{actual}}$) by the STVQ controller are shown in Tab.IV. All slacks are non-negative, which means realtime cores meet their target frame rates in all scenarios. Note that the HD decoder have much more slacks left than the GPU. This is because the pixel-based fine-grained progress notion of the HD decoder facilitates the scheduling job, meanwhile the tile-based progress provided by GPU tile engine is much less accurate. Besides the unpredictable GPU bandwidth makes the QoS-aware scheduling more difficult.
Furthermore, if the frame period is allowed to be late for a few microseconds, which is unnoticeable for human eyes, even lower IPC slowdown rates can be achieved by the STVQ controller.

6.2.3. DRAM Efficiency. As discussed in Section 5, higher DRAM efficiency can be achieved by scheduling more transactions in the same amount of time. Fig. 11 shows the summary of normalized DRAM efficiencies for different test cases. Every DRAM efficiency is normalized through being divided by the efficiency achieved by the STVQ controller on the same test case. As can be seen, the STVQ controller leads to the highest DRAM efficiency in most cases.

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

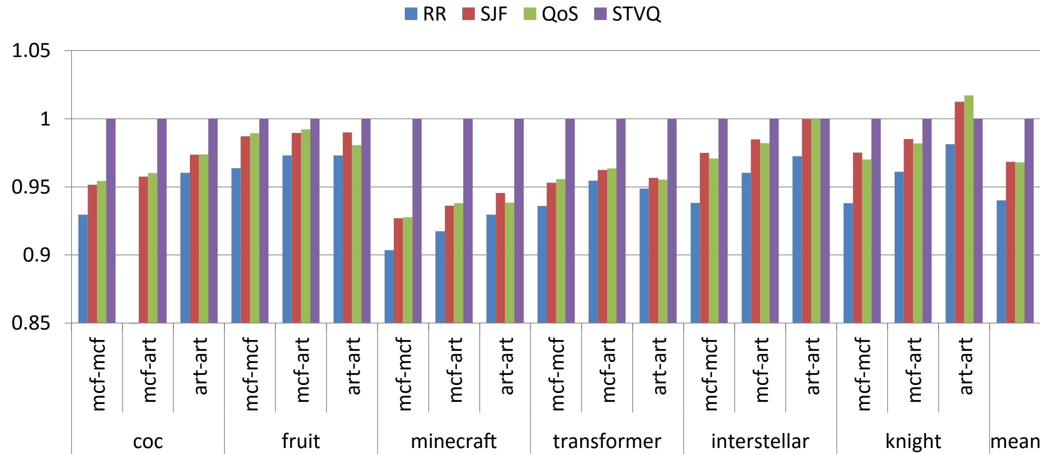


Fig. 11. Summary of normalized DRAM efficiencies during one frame period.

6.3. Batching in the STVQ Memory Controller

In this section, different batching policies will be tested on the STVQ memory controller. According to Section 5.2, batching mechanism helps reduce the average memory latency, especially for traffic flows with high spatial locality. This is achieved by having more row-buffer hits for each row activation. Fig. 12 shows the average numbers of row-buffer hits by real-time traffic flows during one frame period. For each test case, the numbers of row-buffer hits after each row activation are collected under different batching policies. For each batching policy, the average number of row-buffer hits during one frame period is calculated. For comparison, three batching policies are tested: ”no batching” refers to the case using the original scheduling policy in Algorithm 1 without batch formation, and ”batching” means batch formation is adopted in the STVQ controller (Algorithm 2). Two configurations of batching are used: one can have up to 4 transactions in a single batch ($N_{max} = 4$) and the other can have 8 transactions at most ($N_{max} = 8$). Note that the row-buffer hits by CPU traffic are not shown, because batching hardly has an impact on CPU traffic which shows low spatial locality.

Based on Fig. 12, without batch formation the STVQ memory controller achieves reasonable numbers of row-buffer hits which are 4.63 on average across all test cases. In the best case (*mcf-mcf* and *fruit*), there are 5.72 row-buffer hits per activation. By applying batching, more row-buffer hits are achieved. When the maximum batch size is set to 4, the average number of row-buffer hits rises to 5.94. When the batch size limit is adjusted to 8, the number continues to grow until 6.62 which is 43% higher than that when batching is not applied. In the case of *art-art* and *transformer*, the row-buffer hits rise by 66.2% when batching is applied with $N_{max} = 8$.

With more row-buffer hits achieved through batching, real-time traffic observes shorter memory latency. Whereas CPU traffic may suffer from longer memory latency. First, due to the low spatial locality and the sparse traffic pattern, CPU traffic forms into small batches. That means batching cannot reduce much memory latency for the CPU. Second, since real-time traffic can form into large batches easily, CPU traffic often has to wait in the transaction queues until real-time batches have been finished.

Fig. 13 shows the memory latencies collected from both CPU and real-time traffic, with different batching policies. To normalize the latencies, we divide them by the latencies when no batching is applied. Since real-time transactions compose most of

For Peer Review

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60

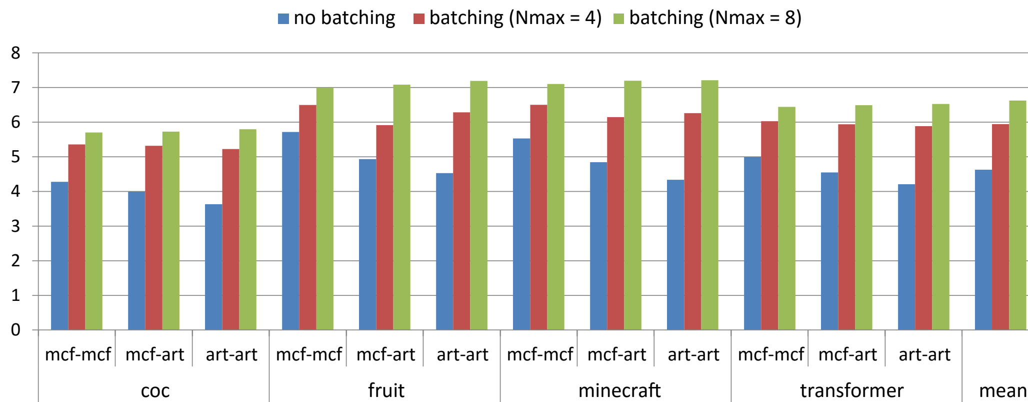


Fig. 12. Average row-buffer hits by realtime traffic during one frame period using the STVQ memory controller with batching.

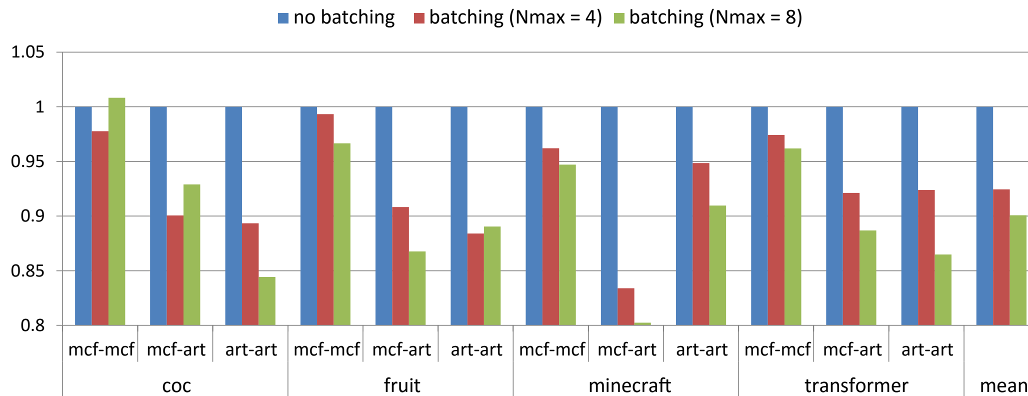


Fig. 13. Normalized memory latencies for all traffic using the STVQ memory controller with batching.

For Review

the memory traffic, more often than not, the reduction on the latency for real-time traffic compensates the increased latency of CPU traffic. Therefore, in most cases, the overall memory latency is decreased. In Fig. 13, the average latency of all test cases sees the reduction of 7.5% when the maximum batch size is set to 4, and 9.9% when the size is set to 8. In the best case (*mcf-art* and *minecraft*), the latency decreases by 19.8% when the size limit is 8.

The CPU IPC under different batching policies are shown in Fig. 14. To be normalized, the IPC numbers are divided by the ones when batching is not applied. Since CPU traffic suffers from more memory latency, IPC is lower than 100% in all test cases when batching is applied. However, the IPC degradations are not severe because the IPC is only lowered by a negligible 0.9% on average. Even in the worst-case (*mcf-mcf* and *transformer*), the IPC is only lowered by 2.3% when the batch size limit is set to 8. But with this small degradation on CPU IPC, batching reduces the memory latency for the overall traffic by up to 19.8% and on average by 9.9%.

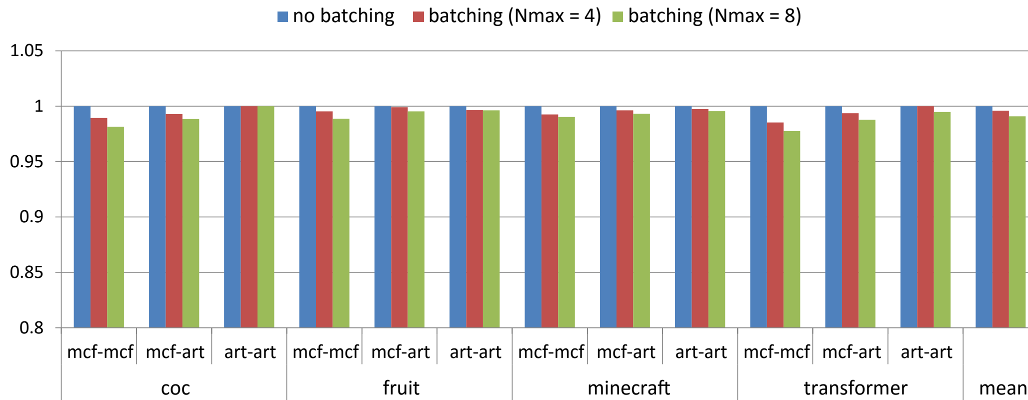


Fig. 14. Normalized CPU IPC using the STVQ memory controller with batching.

6.4. Power and Area

Table V presents the power and area of STVQ in comparison with the two-tier staged memory scheduler [Ausavarungnirun et al. 2012], which has the lowest overhead among previous schedulers. We implemented both memory controllers at the gate level using the TSMC GP 65 nm technology node library. The results are normalized to the results of the two-tier staged memory scheduler. The two memory controllers use input buffers of the same size, as shown in Table I.

As shown in Table V, the STVQ controller saves 8% leakage and 28% area overhead. The savings of the STVQ controller mainly come from the removal of the command queues, whereas adding virtual queues does not increase buffer cost since per-bank queues share the same physical memory. The STVQ memory controller achieves more efficient scheduling with less buffer overhead and high scalability.

Table V. Power and area compared with the staged memory scheduler.

Memory controller	SMS	STVQ
Leakage(normalized)	1	0.92
Area(normalized)	1	0.72

7. CONCLUSIONS

In this work, we show that the existing approaches for heterogeneous MPSoCs memory scheduling based on two-tier queuing architectures have the disadvantage of large queuing delays in the FIFO command stage, which may counteract the benefits of QoS-aware scheduling at the transaction stage. To enable efficacious scheduling, we propose a novel memory architecture with a single-tier virtual queuing system. In addition to our STVQ memory controller, we present a QoS-aware scheduling policy for the CPU and multiple real-time cores. According to our experiments, the STVQ memory controller achieves a lower IPC slowdown rate compared with previous heterogeneous memory schedulers. For the benchmarks evaluated, the reduction in IPC slowdown rate is by up 13.9%, with an average reduction of 6.1%. This reduction is achieved with no frame drops or negative frame slacks for the real-time cores. We also show that the STVQ controller obtains higher DRAM efficiency than previous schedulers. To improve row-buffer hits, we further extend our scheduling policy to support batching. Based on the experimental results, batching increases row-buffer hits by up to 66.2%

For Peer Review

and on average by 43%, which reduces the memory latency for CPU and real-time traffic by up to 19.8% and on average by 9.9%, while the CPU only sees a negligible 0.9% degradation in IPC.

REFERENCES

Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. 2013a. Parallel Frame Rendering: Trading Responsiveness for Energy on a Mobile GPU. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*. IEEE, 83–92.

Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. 2013b. TEAPOT: A Toolset for Evaluating Performance, Power and Image Quality on Mobile Graphics Systems. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*. ACM, 37–46.

Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. 2012. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE, 416–427.

Daniel U. Becker and William J. Dally. 2009. Allocator Implementations for Network-on-chip Routers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*. ACM, 52:1–52:12.

Tungsten Graphics. 2010. Gallium3D. Retrieved Dec. 2014 from http://en.wikipedia.org/wiki/Gallium3D/.

Bruce Jacob, Spencer Ng, and David Wang. 2007. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Javier Jalle, Eduardo Quinones, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. 2014. A Dual-Criticality Memory Controller (DCmc): Proposal and Evaluation of a Space Case Study. In *Proceedings of the Real-Time Systems Symposium (RTSS'14)*. IEEE, 207–217.

Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. 2012. A QoS-aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. ACM, 850–855.

Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. 2011. Minimalist Open-page: A DRAM Page-mode Scheduling Policy for the Many-core Era. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '44)*. ACM, 24–35.

Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010a. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA '10)*. IEEE, 1–12.

Yoongu Kim, M. Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010b. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'00)*. IEEE, 65–76.

Jaekyu Lee, Si Li, Hyesoon Kim, and Sudhakar Yalamanchili. 2013. Design Space Exploration of On-chip Ring Interconnection for a CPU-GPU Heterogeneous Architecture. *J. Parallel Distrib. Comput.* 73, 12 (Dec. 2013), 1525–1538. DOI: http://dx.doi.org/10.1016/j.jpdc.2013.07.014

Asit K. Mishra, Onur Mutlu, and Chita R. Das. 2013. A Heterogeneous Multiple Network-on-chip Design: An Application-aware Approach. In *Proceedings of the 50th Annual Design Automation Conference (DAC '13)*. ACM, 36:1–36:10.

Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, 63–74.

NVIDIA. 2015. Tegra X1. Retrieved Nov. 2015 from http://www.nvidia.com/object/tegra-x1-processor.html.

Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2013. Heterogeneous System Coherence for In-

tegrated CPU-GPU Systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, 457–467.

Qualcomm. 2015. Snapdragon 820. Retrieved Nov. 2015 from https://www.qualcomm.com/products/snapdragon/processors/820.

Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. 2000. Memory Access Scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. ACM, 128–138.

David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Katie Baynes, Aamer Jaleel, and Bruce Jacob. 2005. DRAMSim: A memory-system simulator. In *SIGARCH Computer Architecture News*. 100–107.

Tao Zhang, Cong Xu, Ke Chen, Guangyu Sun, and Yuan Xie. 2014. 3D-SWIFT: A High-performance 3D-stacked Wide IO DRAM. In *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI (GLSVLSI '14)*. ACM, 51–56.

For Peer Review

UNIVERSITY OF CALIFORNIA, SAN DIEGO

BERKELEY • DAVIS • IRVINE • LOS ANGELES • RIVERSIDE • SAN DIEGO • SAN FRANCISCO

SANTA BARBARA • SANTA CRUZ

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING, 0407 LA JOLLA, CALIFORNIA 92093-0407

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60

Dear Editor:

Please find attached our following submission to ACM TODAES:

A Single-Tier Virtual Queuing Memory Controller Architecture for Heterogeneous MPSoCs (authors: Yang Song, Kambiz Samadi, and Bill Lin)

This submission is an extension of our 2016 DAC paper entitled "Single-tier virtual queuing: an efficacious memory controller architecture for MPSoCs with multiple realtime cores". In particular, we significantly extended our work in the following ways:

- We added a new section on memory performance optimization where we discussed new aspects of memory system design, including DRAM efficiency and row-buffer hits optimization. We further extended our scheduling policy to enable batching in the STVQ controller to increase row-buffer hits.

- We provided additional analysis on the test cases for bandwidth allocation by the STVQ controller, as well as the IPC slowdown results.

- We added new experimental results that compare the STVQ controller with previous schedulers with respect to DRAM efficiency, and we showed that the STVQ controller achieves better DRAM efficiency than others.

- We added new experimental results that evaluated the STVQ memory controller with different set-ups on batching, and we showed that the improvements by batching on row-buffer hits and the average memory latency. We concluded that with a negligible degradation on the CPU's IPC, batching reduces the average memory latency by 9.9% on average and 19.8% in the best case.

- We expanded the related work and STVQ memory system sections to provide more background and design details of the STVQ memory controller.

- Finally, we re-wrote several explanations to provide more intuition and we updated different places throughout the paper.

Thank you very for considering our submission.

Sincerely,

Yang Song, Ph.D. candidate
Electrical and Computer Engineering
University of California, San Diego
La Jolla, CA 92093 -0407
E-mail: y6song@eng.ucsd.edu

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

Single-Tier Virtual Queuing: An Efficacious Memory Controller Architecture for MPSoCs with Multiple Realtime Cores

Yang Song[†], Kambiz Samadi[‡], Bill Lin[†]
[†]Electrical and Computer Engineering Department, University of California at San Diego
[‡]Qualcomm Research, San Diego, CA
y6song@ucsd.edu

ABSTRACT

In heterogeneous MPSoCs, memory interference between the CPU and realtime cores is a critical impediment to system performance. Previous memory schedulers adopt the classic two-tier queuing system, but unfortunately the use of two-tier queuing deteriorates the QoS of scheduling policies. In this paper, we propose the Single-Tier Virtual Queuing (STVQ) memory controller for efficacious QoS-aware scheduling. The STVQ memory controller maintains single-tier transaction queues and employs separable allocation for transaction scheduling with high scalability. A multi-source realtime scheduling algorithm is further presented. The STVQ controller achieves up to 13.9% less CPU IPC slowdown than previous schedulers with no frame rate penalty on realtime cores.

1. INTRODUCTION

Heterogeneous MPSoCs have been widely deployed for mobile devices to reduce power while improving system performance. One challenge of integrating heterogeneous cores is to reconcile memory interference among different cores which may have distinct memory traffic patterns. Conventional approaches allocate address space statically or assign different virtual address spaces to heterogeneous cores. Yet the design of memory controllers has not been well explored when heterogeneous cores share the same address space.

A typical realtime core such as a graphic GPU consumes much higher bandwidth than a CPU, because a graphic GPU is capable of executing multiple threads in parallel, which often leads to a large number of memory requests; on the other hand, a realtime core can switch between different threads to hide memory access latency. In contrast, the sparse CPU traffic through the last level cache is much more sensitive to memory stalls. Memory interference happens when the bandwidth-intensive realtime traffic overwhelms the latency-sensitive CPU traffic. Previous works have analyzed heterogeneous memory traffic in details [1, 2, 3, 4, 5].

A classic memory controller for multi-core platforms comprises a two-tier queuing system, including a per-source queuing transaction stage and a per-bank queuing command stage. The two-tier queuing system is a straightforward solution to deliver high memory throughput. Ausavarungnirun et al. [2] were the first to propose the staged memory scheduler (SMS) in the context of integrated CPU-GPU systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '16, June 05-09, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4236-0/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2897937.2898093

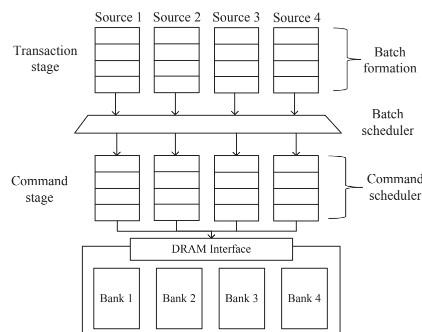


Figure 1: Architecture of a two-tier staged memory scheduler.

(Fig. 1). The proposed SMS architecture follows the two-tier queuing system approach. In SMS, transactions for the same row-buffer in DRAM are formed into batches to increase row-buffer hits. Due to the lower spatial locality, CPU traffic generates shorter batches. By adopting the Shortest Job First (SJF) policy with certain probability, the scheduler prevents CPU traffic from being overwhelmed by the vast GPU traffic. However, the command stage still uses conventional FIFOs, which degrades the QoS improvements from the transaction stage. As a concurrent work, Jeong et al. [3] proposed a realtime QoS-aware scheduling algorithm for the transactions stage, but it also neglects the queuing effects at the command stage.

Effectively, heterogeneous memory scheduling can be depicted as the optimization problem in (1), where L_{cpu} refers to the latency of CPU traffic, $R_{realtime}$ represents the required bandwidths of realtime cores, and FPS (Frames Per Second) is the target frame rate. The CPU can be replaced by any other non-realtime cores. The realtime cores may include a graphic GPU, an HD video decoder, and a display controller, all of which can share the same address space.

$$\begin{aligned} \min \quad & L_{cpu} \\ \text{subject to} \quad & R_{realtime} \geq FPS * framesize. \end{aligned} \tag{1}$$

The drawback of previous works using two-tier queuing systems lies in the unawareness of the queuing delay in the command stage which can be much higher than that in the transaction stage. Performance improvement by the transaction scheduler may not be preserved after the command stage. This problem with two-tier queuing is illustrated in Fig. 2. The figure depicts a common scenario in which the GPU transaction queue has transactions to send but the CPU has no transactions available. As there is no CPU transaction waiting, a QoS-aware scheduler in a two-tier queuing system would send all the GPU transactions queued at the transaction stage to the commmand stage in order to maximize memory throughput. Yet any new incoming CPU transaction going to the same queue would have to wait at

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

the end of the same command queue, which could cause potentially substantial latency to the CPU memory traffic.

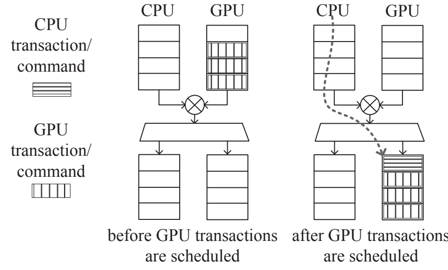


Figure 2: Example of transaction scheduling in a two-tier memory controller causing queuing delay to the CPU.

The problem is further illustrated in Fig. 3 where queuing delays in the transaction and command stages of a two-tier system are shown respectively. For comparison, three scenarios are simulated: one only has a dual-core CPU; another has a dual-core CPU, a graphic GPU, and a display controller, and uses SJF policy for scheduling; the last one also has a dual-core CPU, a graphic GPU, and a display controller, but uses the Round-Robin (RR) policy for scheduling. Even though SJF policy successfully reduces the queuing delay of CPU transactions, the queuing delay of CPU commands remains the same with the result from RR policy, which is much higher than the command delay without realtime cores. In other words the benefit introduced by SJF policy in the first stage is counteracted by the FIFO queues in the second stage. Therefore, we say that a two-tier queuing system is *not efficacious* in the sense that a QoS-aware scheduler may not produce the desired result.

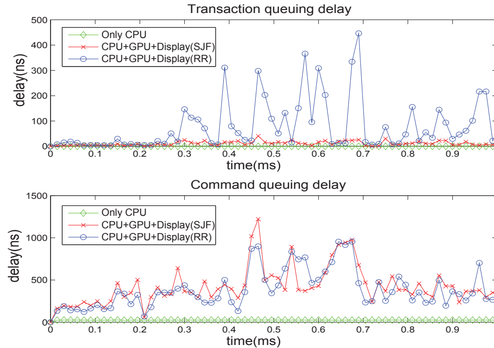


Figure 3: Queuing delays of CPU traffic in the transaction and command stages of a two-tier memory controller. Benchmarks *mcf-art* and *coc* used for CPU and GPU. Experiment setup discussed in Section 5.

In this work, we propose a new memory controller architecture with single-tier virtual queuing for multiple heterogeneous cores. The aim of the STVQ controller is to optimize system performance in (1). The contributions of our work can be summarized as follows.

- **Limitation of two-tier queuing systems for MPSoCs.** We identify the bottleneck of two-tier memory schedulers in delivering good system performance for heterogeneous systems where the CPU and realtime cores have different requirements on latency and bandwidth.
- **Single-Tier Virtual Queuing (STVQ) memory system.** To get rid of the command queuing delay and apply efficacious transaction scheduling, we propose the single-tier virtual queuing memory controller for heterogeneous memory scheduling.
- **Separable allocation for scalable arbitration.** To achieve high scalability, separable allocation is used to implement the selection of memory bank and virtual queue to service, which reduces the complexity of arbitration.
- **Multi-source realtime scheduling.** To modulate traffic flows from the CPU and realtime cores including a graphic GPU, an HD video decoder, and a display controller, we present a QoS-aware scheduling policy for heterogeneous memory traffic from multiple sources.

Our evaluation shows that our proposed STVQ memory controller, together with our proposed realtime scheduling algorithm, can achieve significantly less CPU slowdown under memory interferences from realtime cores. Compared with previously proposed QoS memory controllers, up to 13.9% less IPC slowdown is achieved with 6.1% on average, while realtime cores are able to meet their target frame rates.

The rest of this paper is organized as follows: Section 2 briefly reviews related works. Section 3 describes our proposed single-tier virtual queuing memory controller architecture. Section 4 presents our multi-source realtime scheduling algorithm for the STVQ memory controller. The experimental results and conclusions follow in Sections 5 and 6.

2. RELATED WORK

The First-Ready First Come First Serve (FR-FCFS) policy [6] has been widely used for memory scheduling to deliver high memory throughput, but it assigns more priorities to memory-intensive applications as it prioritizes requests that result in high row-buffer hits. Application-aware scheduling for CPU-only systems has drawn more attention in recent years [7, 8, 9]. ATLAS [7] prioritizes applications with low memory intensity to improve system throughput at the expense of applications with high memory intensity. Thread cluster memory scheduling [8] dynamically clusters applications into low and high memory-intensity clusters and achieves high system performance and fairness simultaneously. Dual-criticality memory controller [9] handles memory contention between realtime and high performance applications by bank separation, assuming no shared memory between different types of applications.

Staged memory scheduler [2] was the first application-aware scheduler proposed for CPU-GPU systems. A two-tier staged memory architecture (Fig. 1) is used to decouple the memory scheduling task into three basic steps, including batch formation, batch scheduling and command scheduling. The batch scheduler switches between two policies which are the Shortest-Job-First (SJF) and Round-Robin (RR). Due to the distinct characteristics of CPU and GPU memory traffic, the SJF policy gives higher priority to the CPU while the RR policy favors the GPU. By adjusting the probability of the SJF policy, the priority of the CPU is modulated to balance performance and fairness for the CPU-GPU system. However, as discussed in Section 1, even though the CPU can achieve sufficient bandwidth through the batch scheduler, there is no guarantee for the latency because the queuing delay in the FIFO command scheduler is uncertain.

In [3] a QoS-aware scheduling policy was proposed to dynamically balance CPU and GPU bandwidth. The proposed policy allocates the minimum bandwidth to the graphic GPU to meet the target FPS. To achieve that, two notions for GPU's frame progress were introduced by (2),

$$P_A = \frac{number\ of\ tiles\ rendered}{total\ number\ of\ tiles}, \quad P_E = \frac{elapsed\ time\ in\ current\ frame}{target\ frame\ time}. \tag{2}$$

Here, P_A is the actual progress of GPU workload in the current frame and P_E is the expected progress. Their QoS policy prioritizes the GPU when P_A is lagging behind P_E. Progress notions can be extended to include other realtime cores such as an HD decoder whose progress metric can be expressed as follows:

$$P_A = \frac{number\ of\ bytes\ (or\ pixels)\ processed}{total\ number\ of\ bytes\ (or\ pixels)}. \tag{3}$$

Nonetheless, this QoS policy focuses on transaction-level scheduling and optimizes CPU bandwidth instead of CPU latency, a more critical performance metric. This may cause performance degradation in a memory scheduler with the two-tier queuing system.

3. SINGLE-TIER VIRTUAL QUEUING MEMORY SYSTEM

A memory system in a single memory channel comprises the physical memory (DRAM devices) and a memory controller. As the bridge between the processors and memory, the memory controller deals with demands from processors (bandwidth, latency, etc.) while following DRAM-access protocols. The objective of our STVQ memory controller is to minimize latency of CPU memory traffic while satisfying frame rate requirements of realtime traffic. Fig. 4 shows an STVQ controller for a DRAM with N memory banks and K independent traffic flows. Four major components reside in the STVQ controller: single-tier virtual queues (VQ), a bank arbiter, a transaction scheduler and per-bank command generators. In this section, we will go through the design details of the proposed single-tier virtual queuing memory system.

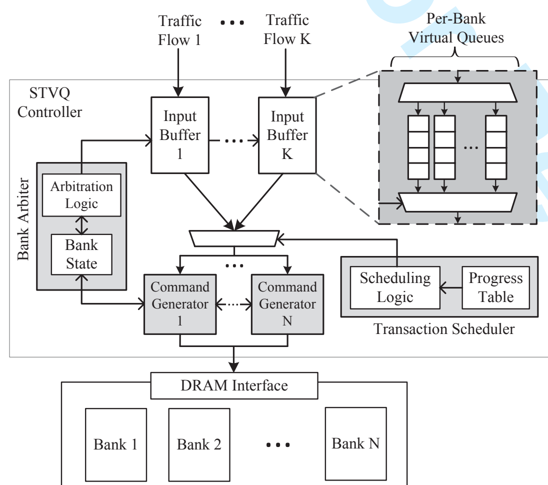


Figure 4: Architecture of the single-tier virtual queuing (STVQ) memory controller.

3.1 Interface for Incoming Traffic Flows

At the input of the memory controller, incoming memory traffic flows from different sources are separated so that QoS-aware scheduling policies can be applied by the transaction scheduler. A realtime core can generate more than one independent traffic flows. In a tile-based graphic GPU, a complete frame is divided into a few tiles which can be rendered in parallel. A typical rendering procedure goes through the geometry shader, rasterizer and fragment shader etc. until the tile cache is flushed into the framebuffer in the external memory. The memory traffic generated during the rendering procedure of current tile, which is mostly input traffic of texture, does not depend on the flushing traffic of previous tiles. Thus GPU memory traffic can be split into rendering and flushing traffic flows. Similarly, an HD video decoder generates reading and writing traffic flows to meet the target frame rate respectively.

3.2 Single-Tier Virtual Queues

In the first stage of the STVQ memory controller, transactions from the same traffic flow are stored in the same input buffer. An input buffer contains a series of per-bank virtual queues. Upon arrival, transactions are sorted into virtual queues based on the destination bank. By using per-bank queues, the STVQ controller is able to schedule transactions to different banks separately.

Since in every clock cycle, the memory controller can only send one command to the DRAM and receive one transaction from each traffic flow, multiple reading or writing does not happen to an input buffer. This allows per-bank virtual queues for the same traffic flow to share one buffer without raising storage cost. In the implementation, virtual queues can be statically allocated, which requires a simple address table to record the start address of each virtual queue inside the input buffers.

3.3 Separable Allocation

To schedule a transaction to DRAM, the STVQ controller selects among per-flow per-bank queues, which is an allocation problem where access to the DRAM interface is granted to one of the virtual queues. Similar to *separable allocators* in on-chip network routers [10], we achieve high scalability by performing allocation with two separable steps: one to choose a memory bank and the other to choose a traffic flow.

In the first step, the bank arbiter assigns DRAM access to one of the memory banks. Meanwhile, virtual queues for this bank are activated in all input buffers and contend for the designated command generator which will convert transactions into DRAM commands. Note that the command generator may receive requests from multiple traffic flows. Therefore, in the second step, the transaction scheduler arbitrates among available traffic flows at the command generator. Only then can the chosen virtual queue send a transaction to the command generator.

By using separable allocation, the STVQ controller solves a simple N:1 or K:1 arbitration problem at a time, where N is the number of banks and K is the number of traffic flows. For example, if there are 4 traffic flows and 8 memory banks, that requires the STVQ controller to perform an 8:1 arbitration to choose a bank, and then a 4:1 arbitration to select a traffic flow. The use of separable allocation significantly reduces the complexity of the arbitration problem.

3.4 Bank Arbiter

In a DRAM device, memory banks operate separately at the same time. For example, bank 1 can be read while bank 2 is being written. Yet since they share the same interface to the memory controller, only one memory bank can receive DRAM command in the same clock cycle. Also, the memory bank is not available for more commands until the current operation has finished. Therefore, the bank arbiter is responsible to find available memory banks and assign one of them with the permission to access DRAM interface.

To achieve this, the arbiter records bank states in a state table and tracks the availability of each memory bank. The arbitration logic selects a winner from available banks based on a certain arbitration policy. In our experiment, the bank arbitration policy does not have notable impact on the performance. Hence we adopt the round-robin policy for simplicity. After arbitration, the bank arbiter sends the bank selection signal to input buffers which activates the corresponding virtual queue for the winning bank and forwards the request to the command generator. Meanwhile the command generator which is designated to the winning bank is also activated.

3.5 Transaction Scheduler

The transaction scheduler arbitrates among requests from different traffic flows at the command generator after bank arbitration. The winning queue schedules a transaction to the command generator afterwards. The traffic flow arbitration and transaction scheduling are performed by the scheduling logic which implements our multi-source realtime scheduling algorithm (discussed in the next section). As input to the scheduler, progress information (2,3) of realtime traffic flows is monitored and recorded by the progress table.

For Peer Review

Table 1: Hardware storage in the STVQ controller with six traffic flows ($K = 6$) and eight DRAM banks ($N = 8$).

Storage	Description	Size
Per-bank VQ	Buffers transactions from a single flow going to a single bank	8 entries
Input buffer	Stores transactions from a single flow	$N \times VQ_Size = 64$ entries
VQ address table	Records each virtual queue's start address	$N \times log_2 Buffer_Size = 48$ bits
Bank state table	Tracks the availability of each memory bank	$N \times 1 = 8$ bits
Progress table	Monitors progresses of realtime traffic flows	$(K-1) \times 2 = 10$ entries
Progress entry	Represents a fractional progress	$2 \times log_2 FrameTime_{max} = 52$ bits

3.6 Command Generators

After bank arbitration and transaction scheduling, only one transaction arrives at the command generator which is designated to the selected memory bank. To execute this memory transaction, the command generator generates commands to operate DRAM while following the DRAM memory-access protocol [11]. During command scheduling, consecutive commands have to be separated by minimum time intervals so that they in turn engage the shared resource in DRAM such as I/O gating and sense amplifiers. The major task of a command generator is to deal with various timing constraints at the command level. Thanks to the command generators, the bank arbiter and transaction scheduler are able to perform allocation without dealing with the intricate DRAM access protocol. Also, per-bank command generators communicate with the bank state table in the bank arbiter to update the bank availability information. In STVQ a command generator is implemented as a state machine which is similar to the command schedulers in prior works [2, 6, 7, 8, 9].

3.7 Hardware Implementation

Virtual Queues: As explained, virtual queues for the same traffic flow share the same input buffer. For an STVQ memory system with N memory banks and K traffic flows, only K buffers are needed. By comparison a two-tier memory system requires $N + K$ buffers because all the queues in Fig. 1 can be active at the same time. Moreover, command queues in the two-tier system can take up lots of storage, because every transaction may generate up to three DRAM commands. By avoiding buffering DRAM commands, storage is spared in the single-tier design.

The storage of an input buffer can be equally allocated to per-bank virtual queues. To record the addresses of N virtual queues in each buffer, N registers are needed. Note that all input buffers can share the same address table for virtual queues.

Bank Arbiter: The logic of bank arbitration is similar to previous memory schedulers. To implement the round-robin policy, the arbiter maintains a register recording the last serviced memory bank. A small state table is used to track availability of memory banks. Each bank has a one-bit flag to indicate its availability.

Transaction Scheduler: The progress table provides the scheduling logic with progress information of realtime traffic. If there are K traffic flows, there should be at most $K - 1$ realtime traffic flows with the other one from the CPU. Each realtime traffic flow has two types of fractional progress, i.e. P_A and P_E (2,3). For each type of progress, two registers are used to store the denominator and the numerator. Specifically, to compute P_E, a register records the target frame time and the other register keeps track of the elapsed time in terms of clock cycles. The computation of P_A for a graphic GPU requires the tile engine in the GPU to output tiling information, including the number of tiles rendered and the total number of tiles in current frame.

Hardware Cost: The hardware parameters of the STVQ memory controller employed in our experiment are configured as follows. We use an STVQ controller working at 1GHz, scheduling memory traffic for 8 memory banks, and receiving 6 traffic flows including two flows (rendering and flushing) from the graphic GPU, two (reading and writing) from the HD decoder, one (reading) from the display controller, and one from the CPU cores. Each virtual queue reserves 8 entries in the input buffer. With 48 virtual queues in total, up to 384 transactions can reside in the controller, which is not expensive compared with the two-tier design [2]. In the progress table, a realtime traffic flow has two entries including P_A and P_E, each of which is a fractional number with two integer components. We assume the lowest admissible frame rate is 23FPS (for movies), which means the maximum frame time contains no more than 64 million clock cycles. Thus a 26-bit register is enough to record the frame time. Two of such registers are used to store a fractional realtime progress. The storage requirements for the STVQ memory controller are summarized in Table 1.

4. MULTI-SOURCE REALTIME SCHEDULING

The scheduling objective of the STVQ controller is to minimize latency for CPU traffic while providing realtime traffic required bandwidth. A realtime virtual queue sends transactions to DRAM when the actual progress falling behind the expected progress. Meanwhile, the CPU receives service as often as possible to reduce queuing delay at the corresponding virtual queue.

Algorithm 1: Transaction Scheduling for CPU and Multiple Realtime Traffic Flows

Input: P_E^i, P_A^i, $ratio^i = \frac{P_A^i}{P_E^i}$ for each realtime queue $i = 1, ..., n$.
1: Find the queue u with the lowest ratio, i.e. $ratio^u = min\{ratio^i | i = 1, ..., n\}$
2: Find the queue v with the highest expected progress, i.e. $P_E^v = max\{P_E^i | i = 1, ..., n\}$
3: **if** $ratio^u \geq 1$ **then**
4: serve CPU transaction queue.
5: **else if** $P_E^v > \rho$ **and** $ratio^v < \beta$ **then**
6: serve realtime transaction queue v.
7: **else if** CPU transaction available **and** $ratio^u > \alpha$ **then**
8: serve CPU transaction queue.
9: **else**
10: serve CPU and realtime queue u in round-robin manner.
11: **end if**

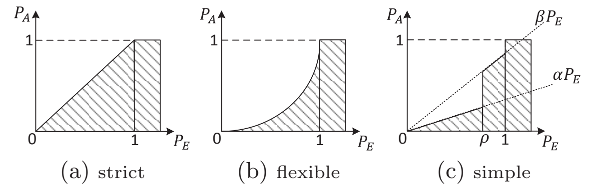


Figure 5: Priority zone of a single realtime traffic flow.

The priority zone of realtime traffic is shown in Fig. 5 where the horizontal axis represents the expected progress (P_E) of a realtime traffic flow and the vertical axis is for the actual progress (P_A). The shaded zone indicates when the realtime traffic obtains priority over CPU traffic, namely, when the actual progress is deemed to be inadequate relative to the expected progress. The expected progress reaches 1 when current frame comes to the end. After that the realtime traffic obtains absolute priority. In a strict implementation as in Fig. 5(a), P_A is forced to keep up with P_E

For Peer Review

Table 3: Benchmark descriptions.

GPU benchmark	Description	FPS
coc	strategy game	50
fruit	action game	60
minecraft	sandbox game	60
transformer	shooting game	60
Video benchmark	Description	FPS
interstellar, knight	4k ultra HD movie trailer	24
CPU benchmark	Description	Bandwidth
art	image recognition	1.06 GB/s
mcf	combinatorial optimization	0.94 GB/s
hmmer	biosequence analysis	0.18 GB/s
sphinx3	speech to text program	0.005 GB/s

Table 2: Simulation parameters.

CPU Parameter	Description
Clock speed	1GHz
L1 cache	32KB private 4-way
L2 cache	1MB shared 16-way
GPU Parameter	Description
Unified shader cores	8
Clock speed	800MHz
DRAM Parameter	Description
Volume	4GB
I/O bus clock	666.67MHz
CL-tRCD-tRP (cycles)	10-10-10
tWTR-tRTP-tWR (cycles)	5-5-10
tRRD-tFAW (cycles)	4-20
Channels-Ranks-Banks	1-1-8

which grows linearly over time. This implementation does not have enough flexibility. When CPU traffic has transactions available, it may be more desirable to serve the CPU while postponing realtime traffic when the current frame will not finish any time soon. Thus in a flexible implementation shown in Fig. 5(b), realtime traffic has less priority than Fig. 5(a) in the beginning so that the CPU has more opportunities to be served. In our experiment, we find it unnecessary to use high order priority functions to achieve good performance. Instead a linear piecewise function $f(P_E)$ (4) is implemented. As shown in Fig. 5(c) realtime traffic obtains priority when $P_A < f(P_E)$. The tunable parameters are empirically determined as $\alpha = 0.5, \beta = 0.95, \rho = 0.95$.

$$f(P_E) = \begin{cases} \alpha P_E & P_E \in [0, \rho) \\ \beta P_E & P_E \in [\rho, 1) \\ 1 & P_E \in [1, \infty) \end{cases} \quad (4)$$

The scheduling algorithm used for multiple realtime traffic flows is shown in Algorithm 1. The realtime traffic achieves the absolute priority when it is supposed to finish the current frame soon ($P_E > \rho$), but its actual progress has not caught up yet ($P_A < \beta P_E$) (Line 6). The CPU obtains priority when it has requests available while the realtime traffic is not too far behind its expected progress, i.e. $P_A > \alpha P_E$ (Line 8), or P_A is leading P_E for all realtime traffic (Line 4). Otherwise, the realtime and CPU traffic share equal priority (Line 10). Note that this algorithm allows realtime traffic to be late for a few cycles which is not noticeable for a graphic application whose frame period is the order of milliseconds.

5. EVALUATION

5.1 Methodology

Our simulation methodology for the STVQ memory controller is based on DRAMSim2 [12], on top of which a heterogeneous system is simulated by a trace-based simulator, including a dual-core CPU, a graphic GPU, an HD decoder, and a display controller. The DRAM model is configured according to the DDR3 SDRAM-1333 specification. The simulation parameters are listed in Table 2. The realtime cores can either include the GPU and display controller, or the HD decoder and display controller, since on a mobile device the display is usually engaged by either the GPU or HD decoder.

For the test traces, we have four CPU applications from SPEC 2000/2006 benchmarks with different memory intensities, including *art*, *mcf*, *hmmer* and *sphinx3*. The GPU traces are extracted from the instrumented softpipe driver in the open-source OpenGL library Gallium3D [13]. Four representative mobile games are selected from different genres. The memory trace for HD decoder is generated from a commercial MP4 parser. More benchmark descriptions can be found in Table 3.

5.2 Bandwidth Allocation

To begin with, we test the STVQ controller with the CPU, GPU and display controller. Benchmarks *mcf* and *art* are used for the CPU and *coc* for the GPU. The bandwidth allocation for the CPU and GPU during one frame period (20 ms) by the STVQ controller is shown in Fig. 6. For comparison two other scenarios are simulated: one only has CPU cores; the other applies SJF for heterogeneous scheduling (in favor of CPU) in a two-tier memory controller.

With no concerns for command queuing delay, the two-tier controller with SJF scheduling allows the GPU to quickly finish all the GPU transactions of current frame and consume most bandwidth in the first 3 ms. At the same time applications on CPU cores hardly make any progress, leading to a delay lasting almost 3 ms. On the other hand, the STVQ controller keeps GPU bandwidth low and steady most of the time to alleviate memory contention against the CPU. In the GPU bandwidth diagram, spikes appear near the end of the frame period when the actual progress of GPU traffic is behind the expected progress and the GPU receives more priority in order to finish current frame in time. The CPU bandwidth diagram shows that the results of the CPU-only system and heterogeneous system with STVQ stay close to each other because the STVQ controller does not cause much latency to CPU traffic. For clarity, the bandwidth of the display is not shown in Fig. 6 since it is very similar to the GPU's.

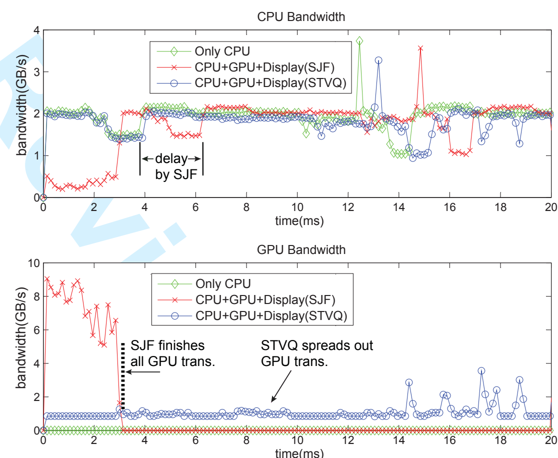


Figure 6: Bandwidth allocation for CPU and GPU.

5.3 IPC Slowdown

Due to memory conflicts, the CPU endures lower IPC when sharing memory with realtime cores, namely, less CPU instructions are executed during the same amount of time. A good memory scheduler reduces IPC slowdown by decreasing memory latency for the CPU. To test STVQ on IPC slowdown reduction, same comparison scenarios are used again including the CPU-only system and the heterogeneous system with a two-tier scheduler using SJF policy.

Fig. 8 shows the committed CPU instructions over time under memory interference from the HD decoder and display controller. Benchmarks *mcf-mcf* and *interstellar* are applied to the CPU and HD decoder. As expected the CPU-only system executes the most instructions over the same period compared with other scenarios. Meanwhile the

For Peer Review

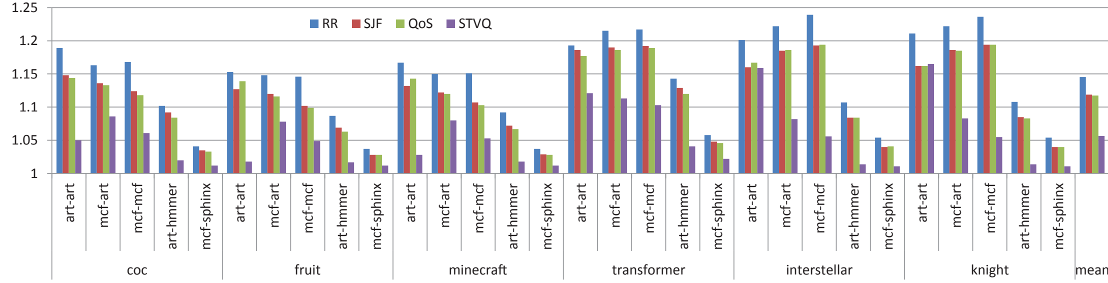


Figure 7: IPC slowdown rate summary (the lower the better).

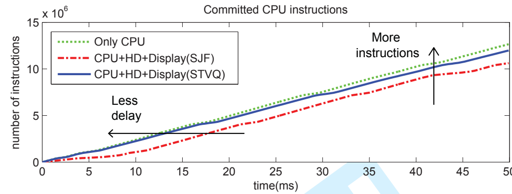


Figure 8: Committed CPU instructions under memory interference from realtime cores.

STVQ controller commits more instructions than the two-tier memory controller. Note that the horizontal distance between two curves shows the time difference after finishing the same number of instructions, which indicates the delay of CPU traffic. The STVQ controller results in much less delay than SJF scheduler.

The IPC slowdown rate is calculated as the ratio of the baseline IPC, when the CPU has exclusive access to memory, to the IPC when the CPU shares memory with realtime cores. Besides the proposed STVQ controller, two-tier memory controller is tested with three scheduling policies including RR policy, SJF policy and an extended version of the QoS policy in [3]. The summary of slowdown rates of all scenarios is in Fig. 7. As expected RR has the worst performance as it prefers bandwidth-intense realtime traffic. Meanwhile SJF and QoS have similar results, because the latency of CPU traffic reduced in the transaction stage is counteracted by the much bigger queuing delay in the command stage. The STVQ controller achieves the best performance except for *art-art* and *knight*. Compared with the lowest slowdown by other schedulers, STVQ further reduces slowdown rate by up to 13.9% (*mcf-mcf* and *knight*) and by 6.1% on average.

Table 4: Summary of frame time slacks (μs).

	art -art	mcf -art	mcf -mcf	art -hmmer	mcf -sphinx
coc	0.0	0.2	0.2	0.0	0.2
fruit	0.6	0.3	0.2	0.4	0.3
minecraft	0.1	0.6	0.4	0.0	0.2
transformer	0.3	0.1	0.1	0.0	0.1
interstellar	18.1	12.0	18.1	5.9	5.9
knight	6.6	13.4	26.8	6.5	6.5

Frame time slacks ($:= \frac{1}{FPS_{target}} - \frac{1}{FPS_{actual}}$) by the STVQ controller are shown in Table 4. All slacks are non-negative, which means the realtime cores meet their target FPS in all test cases.

5.4 Power and Area

The STVQ controller is implemented at the gate level using the TSMC GP 65nm technology node library. Table 5 presents the power and area in comparison with the two-tier SMS controller [2], which has the lowest overhead among previous schedulers. The two controllers use input buffers of the same size, as shown in Table 1. As shown, the STVQ controller saves 8% leakage and 28% area overhead. The savings of STVQ mainly come from the removal of the command queues. STVQ achieves more efficient scheduling with less overhead.

Table 5: Power and area compared with the SMS.

Memory controller	SMS	STVQ
Leakage(normalized)	1	0.92
Area(normalized)	1	0.72

6. CONCLUSIONS

In this work, we show that the existing approaches for heterogeneous memory scheduling using two-tier architecture have the disadvantage of large queuing delay in the FIFO command stage. To enable efficacious scheduling, we propose the single-tier virtual queuing memory architecture. In addition, a QoS-aware scheduling policy for the CPU and multiple realtime cores is presented. The STVQ memory controller achieves lower IPC slowdown rate compared with previous heterogeneous memory schedulers. The reduction of IPC slowdown rate is 6.1% on average and up to 13.9% for the benchmarks. Meanwhile no negative frame slacks are observed in experiment. Finally, the STVQ controller has less hardware overhead than two-tier controllers.

Acknowledgment: We are grateful for the Qualcomm FMA fellowship award.

7. REFERENCES

[1] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous system coherence for integrated CPU-GPU systems. In *ACM/IEEE MICRO*, 2013.

[2] R. Ausavarungnirun, K. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *ACM ISCA*, 2012.

[3] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *ACM/IEEE DAC*, 2012.

[4] J. Lee, S. Li, H. Kim, and S. Yalamanchili. Design space exploration of on-chip ring interconnection for a CPU-GPU heterogeneous architecture. *J. Parallel Distrib. Comput.*, 73(12):1525–1538, Dec. 2013.

[5] A. K. Mishra, O. Mutlu, and C. R. Das. A heterogeneous multiple network-on-chip design: An application-aware approach. In *ACM/IEEE DAC*, 2013.

[6] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ACM ISCA*, 2000.

[7] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *IEEE HPCA*, 2010.

[8] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *ACM/IEEE MICRO*, 2010.

[9] J. Jalle, E. Quinones, J. Abella, L. Fossati, M. Zulianello, and F.J. Cazorla. A dual-criticality memory controller (DCmc): Proposal and evaluation of a space case study. In *IEEE RTSS*, 2014.

[10] D.U. Becker and W.J. Dally. Allocator implementations for network-on-chip routers. In *ACM/IEEE SC*, 2009.

[11] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

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

[13] Gallium3D. http://en.wikipedia.org/wiki/Gallium3D/.

For Peer Review