Fair Queuing Memory Systems

Kyle J. Nesbit, Nidhi Aggarwal, James Laudon[†], James E. Smith

University of Wisconsin – Madison
Department of Electrical and Computer Engr.
{nesbit, nidhia, jes}@ece.wisc.edu

[†]Sun Microsystems, Inc.
james.laudon@sun.com

Abstract

We propose and evaluate a multi-thread memory scheduler that targets high performance CMPs. The proposed memory scheduler is based on concepts originally developed for network fair queuing scheduling algorithms. The memory scheduler is fair and provides Quality of Service (QoS) while improving system performance. On a four processor CMP running workloads containing a mix of applications with a range of memory bandwidth demands, the proposed memory scheduler provides QoS to all of the threads in all of the workloads, improves system performance by an average of 14% (41% in the best case), and reduces the variance in the threads' target memory bandwidth utilization from .2 to .0058.

1. Introduction

Chip multiprocessors (CMPs) will likely form the foundation for a wide range of future computer systems. CMPs allow software threads to share memory system resources with the objectives of efficient resource usage and the accommodation of disparate resource requirements across heterogeneous applications. Of the resources commonly shared, off-chip memory system resources may have the most significant long-term effect on CMP system performance [2]. Unmanaged sharing of these resources can affect system performance in unpredictable ways [10][22] and lead to destructive interference among threads. Destructive interference can result in poor system performance and make OS scheduling policies less effective.

We consider a class of memory scheduling algorithms that service memory requests on a First-Ready First-Come-First-Serve (FR-FCFS) basis [8][17][18] (described in Section 2.2), which has been shown to be a good all-round scheduling algorithm that efficiently optimizes memory bandwidth utilization for single-thread general purpose applications [18][8]. In a multi-thread system, however, naively using FR-FCFS scheduling allows an aggressive thread to severely degrade the performance of other threads with which it is co-scheduled. Figure 1 illustrates this phenomenon – it shows performance measured in instructions per cycle (IPC) and memory latency for the SPEC benchmark *vpr* on a dual processor CMP when *vpr* is running alone, co-scheduled with benchmark *crafty*, and co-scheduled with *art*. The only shared system resource is the SDRAM memory system, i.e., each processor has its own private caches (system details are given in Section 4.1). When running alone, *vpr* consumes a modest *14%* of the system's peak data bus memory bandwidth. When combined with *crafty*, another benchmark with modest memory bandwidth utilization, there is no observable change in *vpr*'s performance. However, when running with benchmark *art, vpr*'s average memory latency goes from *150* cycles (when running alone) to *1070* processor cycles. The increase in latency translates to a *60%* loss of IPC performance. This example illustrates the effect that memory sharing can cause, and it highlights QoS (in the form of performance isolation) as a primary goal of multi-thread memory scheduling policies.

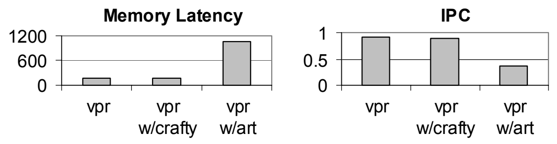

Figure 1: Memory latency and IPC for benchmark *vpr* when it is co-scheduled with *crafty* and with *art*

In this paper, we present a multithread memory scheduler that specifically targets general purpose, high performance CMP systems. The proposed memory scheduler is based on concepts from network fair queuing (FQ) [1][17][20][24]. The FQ memory scheduler models each thread as if it were running in a private virtual time memory system (VTMS). In a thread's VTMS, the memory system's timing characteristics are *time scaled* in proportion to the thread's allocated share of the memory system, e.g., the VTMS CAS delay (t_{CL}) of a thread that is allocated one half of the memory system is doubled. Within this framework, the proposed memory scheduler provides QoS – each hardware thread is offered its allocated share of aggregate memory system bandwidth regardless of the load placed on the memory system from other threads [22].

The proposed memory scheduler is fair – any excess memory bandwidth is distributed evenly to threads that have consumed less excess bandwidth in the past. Furthermore, the proposed memory scheduler improves system performance. On a two processor CMP running workloads that stress the memory system, the proposed FQ memory scheduler provides QoS on *18 out of 19* workloads, improves system performance by *31%* on average (up to *76%)* – and provides good memory system utilization – an average of *92%* of the peak data bus bandwidth. On the one workload where the FQ memory scheduler does not satisfy the QoS objective, the performance is within *6%* of the objective. On a four processor CMP running workloads that have a mix of applications with a range of memory bandwidth demands, the proposed memory scheduler provides QoS to all of the threads in all of the workloads, improves system performance by *14%* on average (up to *41%)*, and reduces the variance in the threads' *target bandwidth utilization* (defined in Section 4.2) from *.2* to *.0058.*

2. Background

2.1. SDRAM Memory Systems

SDRAM memory systems are organized as a set of ranks that consist of independent memory banks. Each bank consists of a two-dimensional array of memory cells. An *activate* command moves a row from the memory array into the row buffer, thereby *opening* the row. Once a row is open, any number of *read* and *write* commands can be issued to transfer data into and out of the row. DDR2 SDRAMs transfer data on both edges of the clock. A *precharge* command closes a row, restores it back to the memory array, and precharges the bank for the next row activation. Throughout this paper we refer to *read* and *write* commands as *CAS commands,* and *activate* and *precharge* commands as *RAS commands*.

DDR2 timing constraints dictate the minimum and maximum time between SDRAM commands. For example, Micron's DDR2-800 timing constraints [14] are summarized in Table 6 in the Evaluation Section. The DDR2 *burst length* (BL) is the number of data bus cycles required to transfer an entire cache line. A more in-depth description of SDRAM memory systems can be found in Cuppu et al. [5].

2.2. Memory Controllers

A *memory controller* acts as the interface between on-chip processors and caches, and off-chip SDRAM, i.e., a memory controller translates memory requests into sequences of SDRAM commands. Figure 2 illustrates the basic structure of a high-performance memory controller [18], which consists of a memory scheduler, transaction buffer, a write buffer, and a read buffer. The transaction buffer holds each memory request's state, e.g., request type and request identifier. The write buffer temporarily holds cache lines being written to memory, and the read buffer temporarily holds cache lines read from memory while they are in transit to the requesting processor's cache.

The *memory scheduler* (shown in the center of Figure 2) is the core of the memory controller. The scheduler reorders and interleaves memory requests in order to optimize memory latency and memory bandwidth utilization.

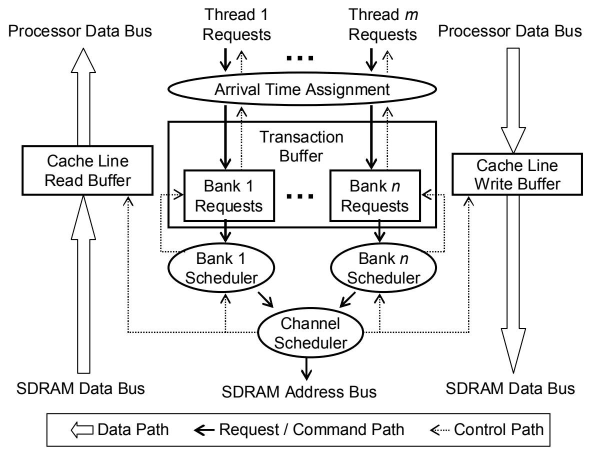

Figure 2: Memory Controller

A high-performance memory scheduler has a logical priority queue and a bank scheduler for each SDRAM bank in the memory system. These queues can be implemented as a single hardware structure, however. The bank scheduler selects the pending request with the highest priority and generates a sequence of SDRAM commands to read (write) the request's data from (to) memory. The bank scheduler also tracks the bank's timing constraints to ensure that the sequence of SDRAM commands conforms to the DDR2 specification. When an SDRAM command is ready (with respect to the bank's timing constraints), the bank scheduler sends the command to the channel scheduler.

The channel scheduler scans the banks' ready commands and issues the command with the highest priority. When a command is issued, the channel scheduler ACKs the appropriate bank scheduler, and the bank scheduler updates its bank state machine appropriately. The channel scheduler also tracks the state of the address bus, data bus, and ranks to ensure there are no channel scheduling conflicts and that no rank timing constraints (e.g. *tRRD* in Table 6) are violated.

Figure 2 illustrates a memory scheduler that employs the commonly used First-Ready First-Come-First-Service (FR-FCFS) memory scheduling algorithm [8][17][18]. FR-FCFS bank and channel schedulers use the same priority policy. The best FR-FCFS policy (as presented by Rixner et al. [19]) has three priority levels: 1) prioritize ready commands over commands that are not ready, 2) prioritize CAS commands over RAS commands, and 3) prioritize the command with the earliest arrival time (i.e. the time memory request arrived at the memory controller). For example, the oldest ready CAS command has the highest priority. Prioritizing first-ready CAS commands exploits already open rows and is essential for utilizing the data bus bandwidth efficiently.

We use a fairly aggressive implementation of the FR-FCFS algorithm as a baseline for performance comparisons. Less aggressive (and lower performing) designs may use separate buffers for read and write memory requests and may employ a FIFO queue to ensure SDRAM timing constraints are met [8]. With such a design, it may be necessary to approximate FR-FCFS [8].

A scheduling policy can have a *closed row policy* or an *open row policy*. A closed row policy closes the row buffer after all pending accesses to the row have been completed, and an open row policy leaves the row buffer open. Throughout this paper, we use a closed row policy since it has been shown to perform better than an open row policy in multiprocessor systems [15].

The FR-FCFS memory scheduler as just described provides good memory system utilization (as our results will show), but it does not provide QoS in a multi-threaded system. There are two main reasons for this. First, FCFS gives unfair priority to threads that have frequent, long bursts of cache misses because a long burst will tend to capture a long sequence of FCFS slots, thereby adding considerably to the latency of other requests that arrive (slightly) later. Second, FR scheduling suffers long priority inversion blocking times due to priority chaining [20]. Priority chaining occurs when a sequence of low priority (later arriving) SDRAM commands prevent high priority (earlier arriving) commands from becoming ready. For example, a stream of row buffer hits will become ready and will be serviced before earlier arriving requests that are to the same bank, but a different row. Overall, FR-FCFS scheduling allows high-demand, bursty threads to prevent other threads from receiving memory system service.

There are relatively few studies of high-performance multi-thread memory schedulers. Zhu et al. [24] study the effects of memory scheduling in simultaneous multithreaded (SMT) processors. Their scheduling techniques primarily focus on scheduling requests based on the threads' occupancies in shared SMT resources, i.e., the reorder buffer (ROB), issue queue, and miss status handling registers (MSHRs). Natarajan et al. [15] study the performance impact of basic memory scheduling techniques (e.g., closed vs. open page policies and in-order vs. out-of-order scheduling) in the context of multiprocessors.

QoS memory schedulers have been proposed for embedded systems and system-on-a-chip (SoC) [7][11][21]. In general, these memory schedulers sacrifice flexibility and memory system utilization in order to achieve hard real-time guarantees. For example, these schedulers often rely on knowing memory access patterns at system design time, which makes them unsuitable for general purpose high-performance computing.

2.3. Network Fair Queuing

Network FQ scheduling algorithms offer guaranteed service to simultaneous network flows over a shared network link [1][16][19][23]. Most FQ algorithms approximate an ideal *general processor sharing (GPS)* server [16]. An ideal GPS server has multiple input buffers, each of which is associated with a different flow and each flow has an allocated service share of the network link. During any time interval when there are backlogged buffers, the ideal GPS algorithm services all backlogged buffers simultaneously in proportion to their corresponding service share.

GPS provides the network flows *Quality of Service* (QoS) and is *fair*. A scheduling algorithm provides *QoS* if each flow receives its allocated service share, regardless of the load placed on the link by other flows. A *fair* network scheduling algorithm distributes excess service in portion to the flows' service shares regardless of the amount of link service a flow has consumed in the past

In an FQ scheduling algorithm a flow i is given a service share ϕ_i expressed as a fraction of the total link capacity. FQ scheduling algorithms often operate within a virtual time framework where the *virtual service time* is equal to the network packet's length L_i^k (expressed in units of link capacity) *time scaled* by the reciprocal of its service share ϕ_i.

Each packet p_i^k (kth packet of ith flow) has a virtual start-time S_i^k and a virtual finish-time F_i^k. The *virtual start-time* (Equation 1) of a packet is the maximum of its virtual arrival time a_i^k and the virtual finish-time of the previous packet. A packet's *virtual finish-time* (Equation 2) is the sum of its virtual start-time and its virtual service time.

[1] $S_i^k = \max \{a_i^k, F_i^{k-1}\}$
[2] $F_i^k = S_i^k + L_i^k / \phi_i$

Network FQ scheduling algorithms use a virtual clock [1][23] to determine the virtual arrival time a_i^k. In general, a virtual clock algorithm advances the virtual clock at a faster rate when fewer network flows are backlogged. A virtual clock is necessary to approximate the GPS fairness policy; the GPS fairness policy is unattainable since a network link cannot simultaneously service more than one packet at a time. An important characteristic of a scheduling algorithm's fairness is the extent to which a flow receiving excess service in a given time period will be penalized in later time period(s) [1][23].

Given the virtual start- and finish-times, there are a number of ways an FQ scheduling algorithm can prioritize packets; each with slightly different QoS and fairness properties [1]. For example, packets can be prioritized earliest virtual start-time first [23] or earliest virtual finish-time first [1][19].

3. Fair Queuing Memory Scheduler

The class of FQ network scheduling algorithms described in Section 2.3 forms the basis of the FQ memory scheduler. In contrast with packet schedulers, however, the FQ memory scheduler must manage multiple, inter-dependent resources, e.g., address buses, data buses, and memory banks. If the FQ scheduler allows a single thread to over utilize a single resource, other threads may be adversely affected.

In an FQ memory scheduler's control registers each hardware thread i is allocated a fraction ϕ_i of the memory system's bandwidth – this allocation could be statically designed-in or could be assigned flexibly by either an OS or a virtual machine monitor (VMM), for example. Allocating a thread a fraction ϕ_i of the memory system is analogous to allocating a thread a private memory system running at ϕ_i of the frequency of the physical memory, i.e., all of the SDRAM timing characteristics are time scaled by the reciprocal of ϕ_i. Therefore, the *FQ memory scheduler's QoS objective* is: a thread i that is allocated a fraction ϕ_i of the memory system bandwidth will run no slower than the same thread on a private memory system running at ϕ_i of the frequency of the shared physical memory system.

To satisfy the QoS objective, the FQ memory scheduler prioritizes memory requests *earliest virtual finish-time first* (VFTF). A virtual finish-time is the virtual time a thread's memory request will finish on the thread's private *virtual time memory system* (VTMS). In order for the FQ memory scheduler to offer a thread QoS, the thread's memory requests must finish in less time than they would have on its VTMS – a memory request's virtual finish-time is its deadline. Scheduling memory requests earliest VFTF is equivalent to earliest deadline first scheduling (EDF) [3]. If the sum of each resource's allocated service shares ϕ_i is less than or equal to one, an ideal EDF schedule will meet the VTMS deadlines [3].

The *FQ memory scheduler's fairness policy* distributes excess memory system bandwidth to the thread that has consumed the least excess memory system bandwidth in the past (relative to its service share). Prioritizing memory requests VFTF also satisfies this fairness policy because a thread whose oldest pending memory request has the earliest virtual finish-time has consumed less excess service (normalized to its share ϕ_i) than any other backlogged thread [1].

The FQ memory scheduler's fairness policy differs from the GPS fairness policy described in Section 2.3. We introduce a fairness policy specifically for multi-thread memory scheduling because a memory system is an integral part of a closed system; in contrast, a network router is assumed to be in an open system. For example, in a typical streaming media network application the bit rate is independent of the delay through the network. In contrast, a general-purpose application's memory request rate is strongly dependent on memory latency, i.e., as the system's memory latency increases the memory request rate decreases. Threads that have consumed more memory system bandwidth in the past have increased the memory system's latency averaged over all threads; these threads should not receive excess bandwidth before threads that have received less excess bandwidth in the past. This fairness policy is validated as part of our evaluation.

3.1. Virtual Time Memory System

A virtual time memory system (VTMS) captures the fundamental timing characteristics of a SDRAM memory system and abstracts away some details. For example, a memory system's address bus and its data bus are modeled as a single resource: a memory channel. This level of abstraction is necessary in order to apply FQ theory and simplify the FQ memory scheduler's hardware (described in Section 3.2).

Table 1 summarizes the VTMS notation and terms. The VTMS notation follows from the network FQ notation used in Section 2.3. In addition, the VTMS notation has a prefix that consists of a resource identifier followed by a dot.

Each memory request m_i^k (k^{th} memory request from the i^{th} thread) to bank j has a bank service virtual start-time $B_j.S_i^k$ and a bank service virtual finish-time $B_j.F_i^k$. The bank service virtual start-time (Equation 3) is the maximum of its virtual arrival time a_i^k (at the memory controller) and the bank service virtual finish-time of the previous request to bank j, i.e., $B_j.F_i^{(k-1)'}$. We use the $^{(k-1)'}$ superscript because memory request m_i^{k-1} may not have been the last request to bank j. The bank service virtual finish-time (Equation 4) is the sum of its virtual start-time and its virtual service time. $B_j.L_i^k$ is m_i^k's actual bank service time. The actual bank service time depends on the state of the bank when the memory request starts its bank service (discussed in Section 3.2).

Table 1: VTMS Notation and Terms

ϕ_i	Thread *i*'s service share
m_i^k	k^{th} memory request from thread *i*
a_i^k	m_i^k's virtual arrival time
$B_j.L_i^k$	m_i^k's bank *j* service
$B_j.S_i^k$	m_i^k's bank *j* service virtual start-time (assuming m_i^k is to bank *j*)
$B_j.F_i^k$	m_i^k's bank *j* service virtual finish-time (assuming m_i^k is to bank *j*)
$B_j.F_i^{(k-1)'}$	Virtual finish-time of the previous request (before m_i^k) to bank *j*
$C.L_i^k$	m_i^k's channel service
$C.S_i^k$	m_i^k's channel service virtual start-time
$C.F_i^k$	m_i^k's channel service virtual finish-time

[3] $B_j.S_i^k = \max \{ a_i^k, B_j.F_i^{(k-1)'} \}$ (from Equation 1)
[4] $B_j.F_i^k = B_j.S_i^k + B_j.L_i^k / \phi_i$ (from Equation 2)

Each memory request m_i^k has a channel service virtual start-time $C.S_i^k$ and a channel service virtual finish-time $C.F_i^k$. In this work we focus on single channel (*C*) memory systems and leave multi-channel memory systems for future work. A memory request virtually arrives at the channel as soon as its bank service is complete, i.e., a request's channel arrival-time is its bank service finish-time $B_j.F_i^k$. Therefore, the channel service virtual start-time (Equation 5) is the maximum of the bank service virtual finish-time and the channel service virtual finish-time of the previous request. The channel service virtual finish-time (Equation 6) is the sum of its virtual start-time and its virtual service time. The channel service virtual finish-time is the memory request's *virtual finish-time*, which is used to prioritize the request.

[5] $C.S_i^k = \max \{ B_j.F_i^k, C.F_i^{k-1} \}$ (from Equation 1)
[6] $C.F_i^k = C.S_i^k + C.L_i^k / \phi_i$ (from Equation 2)

A virtual clock algorithm is unnecessary for the FQ scheduler's desired fairness policy. The FQ memory scheduler uses a real clock. The clock is incremented once per cycle, except during refresh periods. Unlike GPS virtual clock algorithms, a real clock penalizes threads that have consumed more service in the past [19].

3.2. Implementation

The FQ memory scheduler has the same basic structure as the memory scheduler in Section 2.2 Figure 2. The FQ scheduler's priority policy is very similar to FR-FCFS except the FQ scheduler's priority policy prioritizes requests earliest virtual finish-time first. The FQ bank and channel schedulers' priority policy is: 1) prioritize ready commands, 2) prioritize CAS commands, and 3) prioritize commands with the earliest virtual finish-time.

To calculate virtual finish-times, the FQ memory scheduler has a set of VTMS registers and virtual finish-time logic for each supported hardware thread (see Figure 3). A thread's VTMS registers has one finish-time register for each memory bank $B_j.R_i$, one finish-time register for the memory channel $C.R_i$, one service share register ϕ_i, and one register to track the earliest arrival time Ra_i of the pending memory requests. The virtual finish-time registers hold the VTMS resources' last virtual finish-times, i.e. $B_j.R_i = B_j.F_i^{(k-1)'}$ and $C.R_i = C.F_i^{k-1}$. Table 2 summarizes the terms used in this section.

Table 2: VTMS Implementation Terms

$B_j.R_i$	Thread *i* bank *j*'s last virtual finish-time register
$C.R_i$	Thread *i* channel last virtual finish-time register
Ra_i	Thread *i* oldest virtual arrival time
$B_{cmd}.L_i^k$	m_i^k's current SDRAM command bank service
$C_{cmd}.L_i^k$	m_i^k's current SDRAM command channel service

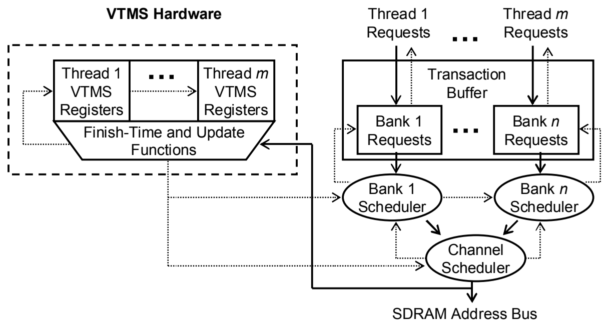

Figure 3: FQ memory scheduler

The virtual finish-time function in terms of VTMS registers is given in Equation 7.

[7] $C.F_i^k = \max \{ \max \{ Ra_i, B_j.R_i \} + B.L_i^k / \phi_i , C.R_i \} + C.L_i^k / \phi_i$
(from Equations 3, 4, 5, and 6)

In contrast with network FQ, a memory request's exact service requirements are not known at the time the request arrives at the memory controller. For example, if a memory request results in an open row hit, then the memory request's bank service requirement ($B.L_i^k$) is t_{CL} , the time to read the data out of the row buffer. If a memory request results in a bank conflict, then the request's bank service requirement is $t_{RP} + t_{RCD} + t_{CL}$, the time to precharge the bank, open the row, and read the data out of the row buffer. A request's bank service requirements are not known until the memory scheduler schedules the request to begin service. Table 3 shows the memory requests' bank service requirements based on the bank state at the time the request begins service. All memory requests require the same channel service ($C.L_i^k = BL/2$).

Table 3: Bank Service $B.L_i^k$ Based on Bank State

Bank State	$B.L_i^k$
Open - bank conflict	$t_{RP} + t_{RCD} + t_{CL}$
Closed	$t_{RCD} + t_{CL}$
Open - row buffer hit	t_{CL}

There are two ways to resolve the bank service discrepancy. The first solution is to assume at arrival time an average bank service requirement for all memory requests, and use the average service requirement to calculate the request's virtual finish-time and update the VTMS finish-time registers. However, this solution is likely to penalize threads that have lower average bank service requirements, e.g., threads with a large number of open row buffer hits. The second solution is to calculate the virtual finish-times of memory requests just before they are scheduled to begin service [1], i.e., when a memory request becomes a thread's oldest first-ready request, and update the VTMS resource finish-time registers after the request has been issued to the memory system. This approach more accurately accounts for the amount of bank service threads actually consume, and consequently, the request's virtual finish-times are more accurate – this is the implementation we evaluate in the Evaluation Section. The disadvantage of this implementation is a thread's oldest first-ready request may not be the thread's oldest memory request, therefore, requests may receive virtual finish-times out-of-order. Despite the reordering, virtual finish-times still respect the bandwidth constraints of a thread's private VTMS.

In our implementation, VTMS finish-time registers are updated whenever a SDRAM command is issued to the memory system. The bank update function is shown in Equation 8. The VTMS channel register $C.R_i$ is only updated when read and write commands are issued, and it is updated after the bank register $B_j.R_i$. The channel register update function is shown in Equation 9.

[8] $B_j.R_i = \max\{a_i^k, B_j.R_i\} + B_{cmd}.L_i^k / \phi_i$
(from Equation 3 and 4)

[9] $C.R_i = \max\{B_j.R_i, C.R_i\} + C_{cmd}.L_i^k / \phi_i$
(from Equation 5 and 6)

The bank $B_{cmd}.L_i^k$ and channel $C_{cmd}.L_i^k$ service times for each type of SDRAM command are summarized in Table 4. The service times for *activates*, *reads*, and *writes* follow from Section 4.1 Table 6. *Precharge* service time accounts for the additional bank service time between issuing an *activate* and a *precharge* that is not accounted for by the *activate*, *read*, or *write* commands.

Table 4: Update Service Times

SDRAM Command	$B_{cmd}.L_i^k$	$C_{cmd}.L_i^k$
Precharge	$t_{RP} + (t_{RAS} - t_{RCD} - t_{CL})$	*n/a*
Activate	t_{RCD}	*n/a*
Read	t_{CL}	*BL/2*
Write	t_{WL}	*BL/2*

The FQ scheduler's hardware is similar to the baseline FR-FCFS scheduler except for the VTMS hardware. VTMS registers can be stored efficiently in SRAM-based register files. After the thread's service share registers are initialized, the bank virtual service times (e.g. $B_j.L_i^k/\phi_i$ and $C.L_i^k/\phi_i$) are constants and do not have to be recomputed each time they are used. Therefore, the update logic and finish-time logic consists of a few adders and muxes. Furthermore, a single copy of the update logic can be shared by all threads because only one thread's registers can be updated in a single cycle.

3.3. Preventing Priority Inversion

Priority inversion blocking time is the time that low priority requests block a higher priority request; it has the potential to significantly degrade the FQ memory scheduler's QoS [20]. With first-ready scheduling, the main source of priority inversion blocking time is bank priority chaining, e.g., when a sequence of low priority row buffer hits prevent a higher priority request from receiving service. To prevent bank priority chaining, a bank scheduler has to select the request with the highest priority and wait for the request's first SDRAM command to become ready; otherwise this command may never become ready. However, scheduling the highest priority request first without accounting for the state of the SDRAM can decrease memory system utilization. For example, a low priority request arrives at a memory bank and activates its row. Then, a cycle after the low the row is activated, a higher priority request arrives at the same bank. A bank priority policy that schedules the request with the highest priority and waits for its first SDRAM command to become ready will wait t_{RAS} *-1* cycles before the higher priority request's *precharge* command becomes ready, i.e., DDR2 timing constraints (see Table 6) require t_{RAS} cycles between issuing an *activate* and a *precharge*. During this time the low priority request could have completed its memory transaction without adding to the latency of the higher priority request.

In general, a memory scheduler that reduces priority inversion blocking time decreases data bus utilization. Consequently, we give the FQ bank scheduler a configurable bound x on priority inversion blocking time. The FQ bank scheduler's priority policy is as follows. When the bank is closed, during the first x memory cycles after an *activate* command, the bank scheduler's priority policy is the same policy presented in the previous section. After the bank has been active for x cycles, the bank scheduler selects the request with the earliest virtual finish-time and waits for its first SDRAM command to become ready. In this paper, we use t_{RAS} for the value of x. This is a tight bound on priority inversion blocking time, which offers better QoS, but may decrease data bus utilization.

The FQ memory scheduler offers threads a high degree of QoS. However, strictly speaking, the FQ memory scheduler does not *guarantee* QoS since a thread's memory resource requirements may change when the thread's memory requests are interleaved with another thread's memory requests, e.g., a memory request that would have been a row buffer hit on a private memory system may cause a bank conflict on a shared memory system. Providing strict QoS guarantees would require pessimistic assumptions that are unsuitable and unnecessary for high-performance computing [7][11][21].

4. Evaluation

4.1. Methodology

The FQ memory scheduler allows arbitrary fractions of memory system bandwidth to be allocated to an individual processor or a cluster of processors. However, in this paper our target system is a general purpose desktop or laptop with off-the-shelf software. In this scenario, processors on the CMP are *statically* allocated an *equal* share ϕ_i of the memory system, e.g., in a two processor CMP each processor is allocated $\phi_i = ½$ of each memory resource. We show that the FQ memory scheduler offers threads a high degree of QoS and that QoS in the form of performance isolation is essential. Providing QoS in other scenarios (e.g. real-time multimedia applications, servers with scientific or commercial multithreaded workloads, logical partitioning of servers) are topics for future research.

We use a detailed structural simulator that was developed at IBM Research. The simulator has a structure similar to ASIM [6], and the model is defined at an abstraction level slightly higher than a latch-level model. Our model partitions processor logic into pipelines – each pipeline consists of multiple pipeline stages. The simulator models the finite capacity and bandwidth of each data flow path, buffer, and bus in a typical processor. The model's default configuration is of a single processor IBM 970 system. In its default configuration, the model's performance projections have been validated to be ± 5% of the 970 design group's latch-level processor model. In this paper we use an alternative processor configuration to avoid some 970-specific design constraints. The primary differences are: 1) a monolithic reorder buffer instead of the 970's instruction dispatch groupings, 2) a unified issue queue instead of the 970's distributed issue queue, and 3) a cache configuration that is more representative of modern desktop and laptop processors than the 970's cache configuration For example, we use 64 byte line sizes – the 970's 128 byte lines increase memory bandwidth contention and amplify the effects of the FQ scheduler – 64 byte lines result in better baseline performance.

The simulator has a cycle accurate model of an on-chip memory controller attached to a DDR2-800 memory system. The memory controller maps physical addresses to ranks and banks using an XOR address mapping [12]. The system configurations are in Table 5. The SDRAM memory system configuration was chosen to be representative of most modern desktop and laptop systems. The DDR2-800 timing constraints [14] are in Table 6.

Table 5: 4 GHz Processor – System Configuration

Issue Buffer	64 entries
Issue Width	8 units (2 FXU, 2 LSU, 2 FPU, 1 BRU, 1 CRU)
Reorder Buffer	128 entries
Load / Store Queues	32 entry load reorder queue, 32 entry store reorder queue
I-Cache	32KB private, 4-ways, 64 byte lines, 2 cycle latency, 8 MSHRs
D-Cache	32KB private, 4-ways, 64 byte lines, 2 cycle latency, 16 MSHRs
L2 Cache	512KB private, 8-ways, 64 byte lines, 12 cycle latency, 16 store merge buffer entries, 32 transaction buffer entries
Memory Controller	16 transaction buffer entries per thread, 8 write buffer entries per thread, closed page policy
SDRAM Channels	1 channel
SDRAM Ranks	1 rank
SDRAM Banks	8 banks

Table 6: Micron DDR2-800 timing constraints (measured in DRAM address bus cycles)

t_{RCD}	*Activate* to *read*	5 cycles
t_{CL}	*Read* to data bus valid	5 cycles
t_{WL}	*Write* to data bus valid	4 cycles
t_{CCD}	CAS to CAS (CAS is a *read* or a *write*)	2 cycles
t_{WTR}	*Write* to *read*	3 cycles
t_{WR}	Internal *write* to *precharge*	6 cycles
t_{RTP}	Internal *read* to *precharge*	3 cycles
t_{RP}	*Precharge* to *activate*	5 cycles
t_{RRD}	*Activate* to *activate* (different banks)	3 cycles
t_{RAS}	*Activate* to *precharge*	18 cycles
t_{RC}	*Activate* to *activate* (same bank)	22 cycles
BL/2	Burst length (Cache Line Size / 64 bits)	4 cycles
tRFC	*Refresth* to *activate*	51 cycles
tRFC	Max *refresh* to *refresh*	28,000 cycles

To isolate the effects of sharing main memory, the SDRAM memory system is the only shared resource in the system (i.e., the processor cores have private caches). However, in practice, the FQ memory scheduler can be implemented with shared caches and would likely work well with fair sharing / QoS caches; see Kim et al. [10]. The memory controller's transaction buffer and write buffer are statically partitioned. Each thread is allocated 16 transaction buffer entries, and 8 write buffer entries. The memory controller NACKs memory requests from a thread when that thread's buffer entries are full, thus applying back pressure to that thread independent of the other threads on the CMP. A more flexible partitioning of memory controller's buffers is possible and is a topic for future research.

We use the SPEC 2000 benchmark suite because it is the best available source of heterogeneous applications. We use twenty 100 million instruction SPEC benchmark sampled traces that have been verified to be statistically representative of the entire SPEC application [8].

We use data bus utilization as a measure of a thread's "aggressiveness" in the memory system, although we acknowledge that there are other factors that might contribute to aggressiveness (e.g., average burst size, row hits, bank parallelism, etc.). Figure 4 illustrates the data bus utilization of the twenty SPEC traces when running alone on a single processor with the FR-FCFS memory scheduler. Throughout the results section, benchmarks are ordered by their data bus utilization; the highest utilization (the most aggressive) is on the left.

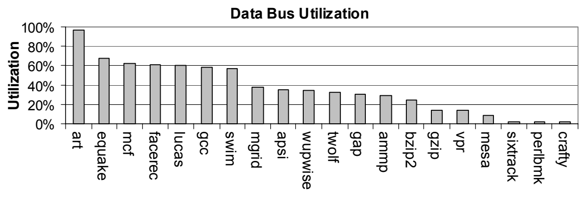

Figure 4: Data Bus Utilization of individual benchmarks running alone

4.2. Results

The first set of results illustrates the effect of co-scheduling a *subject* thread with a very aggressive *background* thread. This set of experiments stresses the FQ memory scheduler's ability to isolate the performance of the subject thread from the background thread. Based on data given in Figure 4 we chose *art*, the most aggressive benchmark, to be the background thread and simulated *art* with every other benchmark as the subject thread. Figure 5 shows the normalized IPC, average memory read latency, and data bus utilization of the subject thread listed on the x-axis. *FR-FCFS* is the first-ready first come first serve (earliest arrival time first) scheduling algorithm. *FR-VFTF* is the first-ready virtual finish-time first scheduling algorithm; this configuration uses first-ready scheduling *without* the FQ bank scheduling algorithm described in Section 3.3. *FQ-VFTF* is the FQ memory scheduler. It prioritizes requests VFTF and uses the FQ bank scheduling algorithm to prevent priority chaining. IPC is normalized with respect to the same benchmark running alone on a private memory system with the timing constraints in Table 6 *time scaled by a factor of two,* i.e., $1 / \phi_i$. We refer to this system as the *baseline* system. An ideal QoS memory scheduler would have no benchmark with a normalized IPC less than one. Data bus utilization is measured with respect to peak data bus bandwidth.

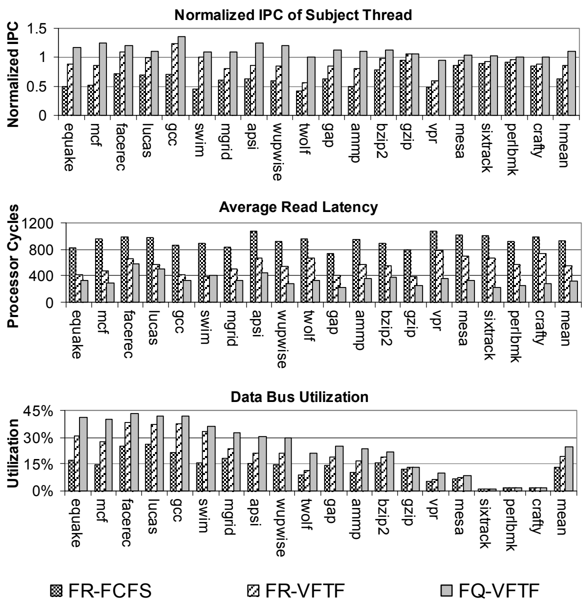

Figure 5: A subject thread's normalized IPC (top), average read latency (middle), and data bus utilization (bottom)

Figure 5 illustrates that an aggressive background thread can have a significant affect on a less aggressive subject thread. With the FR-FCFS memory scheduler, the IPC of the subject thread is often less than one half the IPC of the baseline system, and the harmonic mean of normalized IPC is *.62*. The decrease in IPC is caused by the dramatic increase in the subject thread's average memory read latency (on average *930* processor cycles). In contrast, the memory system's unloaded read latency is *180* cycles. This result supports the importance of memory system bandwidth and the need to control memory interference.

In Figure 5, the FR-VFTF scheduler's results show that the VFTF priority algorithm is an important component of the FQ memory scheduler, but by itself it is not enough to provide QoS. With the FR-VFTF scheduler, the normalized IPC is less than one (i.e., below the QoS objective) for *14 out of 19* workloads, and in the case of *twolf* and *vpr*, normalized IPC is less than *.6*.

The FQ memory scheduler (FQ-VFTF) provides a high degree of QoS, i.e., it offers the subject thread QoS on *18 out of 19* workloads. The one bencmark, *vpr,* where the FQ memory scheduler does not provide QoS has a normalized IPC less than one because it has little memory parallelism. This characteristic makes *vpr* very sensitive to the memory system's preemption latency, i.e., the time to finish or preempt a lower priority request in order to service a higher priority request. Despite this, *vpr*'s normalized IPC with the FQ memory scheduler is *.94*, which is much better than *vpr's* normalized IPC with FR-FCFS (*.48*) or FR-VFTF (*.58*). In addition, with the FQ memory scheduler, the data bus utilization of the subject thread tracks the data bus utilization of the same application when running alone on its own memory system (Figure 4). Comparing FR-VFTF with the FQ memory scheduler (FQ-VFTF) illustrates the effects of first-ready priority chaining and the FQ bank scheduling algorithm's effectiveness. The FR-VFTF scheduler has no mechanism to control priority chaining. Its average normalized IPC *.87.* The FQ memory scheduler (FQ-VFTF) scheduler uses the FQ bank scheduling algorithm (Section 3.4) to limit the effects of priority chaining and its average normalized IPC is *1.10,* an improvement of *26%*.

Figure 6 shows the background thread's normalized IPC; the subject thread's benchmark is listed on the x-axis. This figure illustrates that the FQ memory scheduler provides the background thread QoS and efficiently distributes excess memory service to the background thread. Starting from the right, the first six subject threads demand more than half of the memory system bandwidth (see Figure 4). For these workloads the actual memory system bandwidth is divided evenly. Therefore, for the first six workloads the normalized IPC of the background thread is very close to one. Beyond the first six benchmarks (from the left), the normalized IPC of the background thread increases gradually as the background thread receives more excess service.

In addition to providing QoS, the FQ scheduler eliminates negative memory interference, and this significantly improves overall system performance. The top graph in Figure 7 shows the performance improvement of the FR-VFTF and FQ memory schedulers when compared with FR-FCFS as the baseline. The performance metric used in Figure 7 is the harmonic mean of the co-scheduled threads' normalized IPCs [13]. Another important memory scheduler performance metric is memory system throughput (measured as data bus and bank utilization by the bottom two graphs of Figure 7). Because memory system bandwidth is a critical resource for future CMPs, it is important that the memory scheduler use it efficiently.

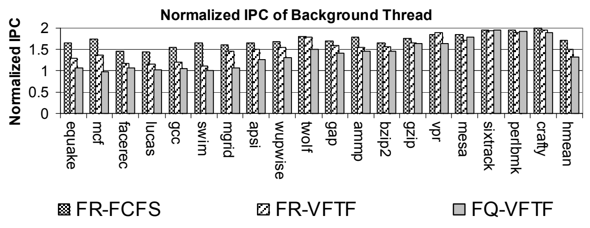

Figure 6: Normalized IPC of background thread

The performance improvement graph in Figure 7 illustrates that an overly aggressive application can significantly degrade the performance of the entire system, and, in such cases, providing QoS in the form of performance isolation increases system performance. For this experiment, the FQ memory scheduler (FQ-VFTF) has an average performance improvement of *31%* (up to *76%*).

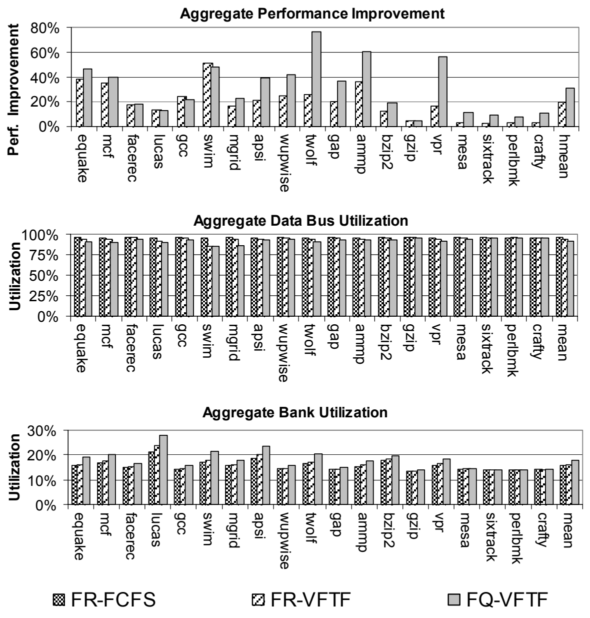

Figure 7: Aggregate performance improvement (top), aggregate data bus utilization (middle), and aggregate bank utilization (bottom)

The data bus utilization graph in Figure 7 show that the FR-FCFS effectively optimizes data bus utilization. This result supports our decision to use FR-FCFS as the baseline and as the basis of the FQ memory scheduler. Overall, FR-VFTF and FQ-VFTF also have good data bus utilization. On average their data bus utilizations are *94%* and *92%*, respectively. However, there are some benchmarks (e.g., *swim* and *mgrid*) that have a considerable decrease in data bus utilization – up to *10%*. The decrease in data bus utilization is caused by an increase in bank conflicts, which is illustrated by the increase in aggregate bank utilization. The increase in bank utilization is caused partly by the VFTF priority algorithm and partly by the FQ bank scheduling algorithm. In general, an increase in bank utilization is an unavoidable side effect of offering threads QoS. This result emphasizes the importance of bank bandwidth in future CMP memory systems.

The second set of results shows the benefits of FQ scheduling in a typical desktop scenario – a four CMP system running heterogeneous applications that have a range of memory behaviors. To construct the workloads, we combined every fourth benchmark (in order of data bus utilization), e.g. the first workload consists of the 1st, 5th, 9th, and 13th benchmarks (*art*, *lucas*, *apsi*, and *ammp*). We chose to exclude the last four benchmarks since they have very low memory system utilization; i.e., *sixtrack*, *perlbmk*, and *crafty* utilize less than 2% of the memory bandwidth.

Figure 8 shows the normalized IPC and data bus utilization of the four processor workloads. The workloads are ordered by memory system demand with the most demanding workload on the left. The benchmarks within a workload are ordered by the threads' aggressiveness with the most aggressive thread on the left. The baseline for this experiment is a single processor system with a private memory system *time scaled by a factor of four.* Normalized IPC is normalized to the benchmark running alone on the baseline system.

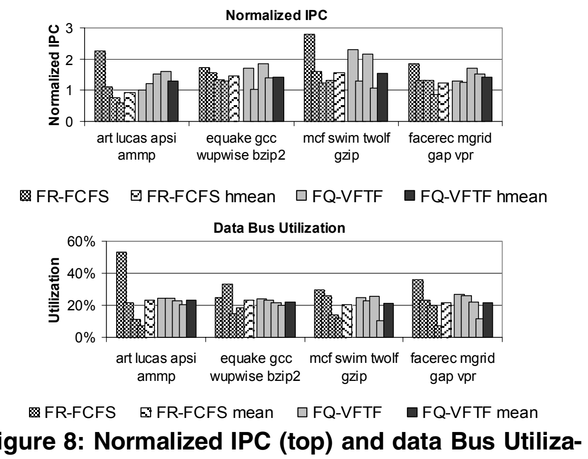

Figure 8: Normalized IPC (top) and data Bus Utilization of individual threads in a four processor CMP system (bottom)

The leftmost workload in Figure 8 (*art, lucas, apsi,* and *ammp*) has the highest aggregate memory bandwidth demand, each thread demands over 25% of the memory bandwidth, and this workload's constituent benchmarks have the widest range of memory demands (see Figure 4). With the FR-FCFS memory scheduler, the most aggressive thread (*art*) receives the most memory system service and has the greatest normalized IPC, while the two least aggressive benchmarks (*gap* and *vpr*) receive the least memory system service and have a normalized IPC less than one (below the QoS objective). With the FQ memory scheduler (FQ-VFTF) the opposite is true; the least aggressive thread has the greatest normalized IPC, and all threads have a normalized IPC greater than one. The FQ memory scheduler's (FQ-VFTF) data bus bandwidth distribution (bottom graph of Figure 8) for this workload is nearly ideal; the ideal distribution is a uniform distribution. Even with the FQ scheduler, more aggressive threads still tend to get slightly more memory bandwidth. With general purpose processors this appears to be unavoidable. Threads that are aggressive with respect to the memory system tend to have more memory level parallelism, which allows them to generate more memory requests when given less memory service. This result supports our choice of fairness, i.e., giving excess memory service to threads that have been less aggressive in the past.

The effect of the FQ memory scheduler on the remaining three workloads is not as clear as the first workload, mainly because the benchmarks within the last three workloads have a narrower range of memory bandwidth demands and not all of the benchmarks demand their full share of the memory system (i.e., *gzip* and *vpr*). Two important trends for the last three workloads are: 1) the memory scheduler has a significant affect on the individual benchmarks' performances and 2) the FQ memory scheduler more evenly distributes memory bandwidth when compared with FR-FCFS.

The FQ memory scheduler's performance improvement for first, second, third, and fourth workloads (from the left) is *41%*, *-2%*, *-2%*, and *14%* respectively. On the second and third workloads the overall system performance is virtually unchanged (we consider ± 2% to be within the error margin of the experiment), however, the normalized IPCs of the individual threads are significantly affected. Overall, the FQ memory scheduler improves the aggregate performance of all the workloads by *14%*.

To provide further insight we compare each thread's actual data bus utilization (from Figure 8) with its *target bus utilization*. A thread's target data bus utilization is the smaller of 1) its data bus utilization when running alone (solo) on the CMP and 2) the sum of its allocated service share plus its fair share of excess memory bandwidth. For a four processor CMP a thread's target data bus utilization is: min{ *<solo data bus utilization>* , *25%* + *<fair-share of excess bandwidth>* }. A threads' *solo data bus utilization* is taken from Figure 4. A thread's *fair-share of excess bandwidth* is determined by incrementally adding equal portions of excess service to each thread that demands service until all excess service is allocated or there are no threads that demand more service (when a thread's target data bus utilization equals its solo data bus utilization). On average, this is the fair-share to which the FQ memory scheduler's fairness policy will ideally converge.

Figure 9 shows the normalized read latency versus its normalized data bus utilization for all the threads in Figure 8. Read latency is normalized to the read latency when a benchmark is run alone and data bus utilization is normalized to the application's target data bus utilization – both are derived from the results collected for Figure 4. With an ideal memory scheduler, all threads would have a normalized data bus utilization of one. The vertical line in Figure 9 is a visual aid to illustrate the *ideal*.

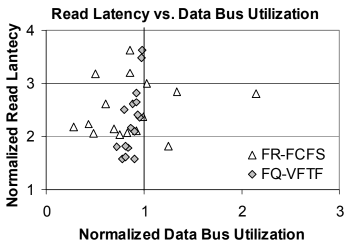

Figure 9: Normalized Latency vs. Normalized Data Bus Utilization of individual threads

The FR-FCFS memory scheduler's normalized latency and normalized data bus utilization show no discernable trend. The average normalized data bus utilization is *.88*, has a range from *.28* to *2.1,* and a variance of *.20.*

In contrast, the FQ memory scheduler's (FQ-VFTF) normalized data bus utilization is clustered slightly to the left of the *ideal* vertical line at one. For FQ-VFTF, the average normalized data bus utilization is *.88*, has a range from *.73* to *.98* and a variance of *.0058.*

The average normalized data bus utilization is a metric of the scheduler and multithread workload's *data bus efficiency*, i.e., it is a workload weighted average of the data bus utilization. FR-FCFS and FQ-VFTF have the same data bus efficiency (*.88*), which is approximately the same as the average data bus utilization in Figure 8. A data bus efficiency of less than one causes the FQ-VFTF scheduler's cluster of normalized data bus utilization to be slightly to the left of the ideal vertical line.

Another important observation from Figure 9 is the increasing relationship between the FQ memory scheduler's (FQ-VFTF) normalized data bus utilization and normalized read latency. This result illustrates again that aggressive threads tend to get slightly more bandwidth than less aggressive threads. Furthermore, the increasing trend illustrates that less aggressive threads which receive less memory bandwidth have a lower normalized latency, e.g., the cluster around *.8* normalized data bus utilization and *1.7* normalized read latency. In contrast, more aggressive threads have a much higher normalized read latency, e.g., the cluster around *.98* normalized data bus utilization and *3.5* normalized read latency. This result further supports the FQ scheduler's fairness policy. A formal validation of the FQ scheduler's fairness policy is a topic for future work.

5. Summary and Conclusions

Uncontrolled sharing of memory system resources can cause destructive interference between threads. We have shown that the effect on performance due to memory sharing alone can be significant and a thread with aggressive memory system usage can starve other threads, which can make the OS scheduling policies less effective.

Current memory scheduling policies that are optimized for single-thread environments are not well suited for multi-thread workloads. FR-FCFS, the best existing single thread technique, when used in a CMP, performs poorly because its FCFS priority algorithm gives unfair advantage to threads with frequent long bursts of memory access patterns and its FR scheduling policy is subject to long priority inversion blocking times.

We proposed the Fair Queuing memory scheduler. The basis of the FQ memory scheduler is fair queuing algorithms from computer networking research. The QoS objective of the FQ scheduler is to ensure each thread receives its allocated fraction of the memory system regardless of the load placed on the memory system by other threads. In addition, the FQ scheduler fairly distributes excess memory system service to threads that have consumed less service in the past.

We showed the FQ memory scheduling algorithm offers threads QoS and can be incorporated in existing memory schedulers with low implementation complexity. Furthermore, we showed that reducing the destructive interference due to memory system sharing improves system performance. On a four processor CMP running workloads that have a mix of applications with a range of memory bandwidth demands, the proposed memory scheduler provides QoS to all of the threads in all of the workloads, improves system performance by *14%* on average (up to *41%*) and reduces the variance in a thread's normalized bandwidth utilization from *.2* to *.0058*.

6. Acknowledgements

This work was supported by an IBM Fellowship and by equipment donations and financial support from Intel. The first author would like to thank Professor Parmesh Ramanathan for his clarity and insights in teaching real-time systems and his direction during the formative stages of this research. We would also like to thank Ravi Nair and Dan Prener of IBM for their advice and support during the development of simulation infrastructure.

7. References

[1] Bennett, J. C. and Zhang, H. Hierarchical packet fair queueing algorithms. In *Proc. on Apps. Tech., Arch., and Protocols for Comp. Comm.* (Palo Alto, Ca., Aug. 28 - 30, 1996). SIGCOMM '96. 143-156.

[2] Burger, D., Goodman, J. R., and Kägi, A. Memory bandwidth limitations of future microprocessors. In *Proc. of the 23rdl Intl. Symp. on Comp. Arch.* (Philadelphia, PA May 22 - 24, 1996). ISCA '96. 78-89.

[3] Chetto, H. and Chetto, M. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Trans. on Software Eng.* 15, 10 (Oct. 1989), 1261-1269.

[4] Cuppu, V. and Jacob, B. Concurrency, latency, or system overhead: which has the largest impact on uniprocessor DRAM-system performance. In *Proc. of the 28th Intl. Symp. on Comp. Arch.* (Göteborg, Sweden, June 30 - July 04, 2001). ISCA '01. 62-71.

[5] Cuppu, V., Jacob, B., Davis, B., and Mudge, T. A performance comparison of contemporary DRAM architectures. In *Proc. of the 26th Intl. Symp. on Comp. Arch.* (Atlanta, Georgia, May 01 - 04, 1999). ISCA '99. 222-233.

[6] Emer J., e.t al. Asim: A Performance Model Framework. *Computer* 35, 2 (Feb. 2002), 68-76.

[7] Hofstee P., Nair R., and Wellman J.D. Token Based DMA, United States Patent 6820142. (Nov. 16, 2004).

[8] Hur, I. and Lin, C. Adaptive History-Based Memory Schedulers. In *Proc. of the 37th Intl. Symp. on Microarchitecture* (Portland, Oregon, Dec. 04 - 08, 2004). 343-354.

[9] Iyengar, V. S., Trevillyan, L. H., and Bose, P. 1996. Representative Traces for Processor Models with Infinite Cache. In *Proceedings of the 2nd Symp. on High-Perf. Comp. Arch..* (Feb. 03 - 07, 1996). HPCA '96.

[10] Kim, S., Chandra, D., and Solihin, Y. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proc. of the 13th Intl.l Conf. on Parallel Arch. and Compilation Techniques* (Sept. 29 – Oct. 03, 2004). PACT '04. 111-122.

[11] Lee K., Lin T., Jen C. An Efficient Quality-Aware Memory Controller for Multimedia Platform SoC. *IEEE Trans. on Circuits and Systems for Video Technology*, Volume 15, Issue 5, (May 2005) 620 – 633.

[12] Lin, W. et al. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In *Proc. of the 7th Intl. Symp. on High-Perf. Comp. Arch.* (Jan. 20 - 24, 2001). HPCA '01. 301.

[13] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *Proc. of the Intl. Symp. on Perf. Analysis of Systems and Software*, (Jan. 2001): 164-171,.

[14] Micron. 1Gb DDR2 SDRAM Component: MT47H128M8B7-25E, June 2006.

[15] Natarajan, C., Christenson, B., and Briggs, F. A study of performance impact of memory controller features in multi-processor server environment. *In Proc. of the 3rd Workshop on Memory Perf. Issues* (Munich, Germany, June 20 - 20, 2004). WMPI '04. 80-87.

[16] Parekh, A. K. and Gallager, R. G. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Trans. Networks* 1, 3 (Jun. 1993), 344-357.

[17] Rixner, S., Dally, W J., Kapasi, U. J., Mattson, P., and Owens, J. D. Memory access scheduling. In *Proc. of the 27th Intl. Symp. on Comp. Arch.* (Vancouver, British Columbia, Canada). ISCA '00. 128-138.

[18] Rixner, S. Memory Controller Optimizations for Web Servers. In *Proc. of the 37th Intl. Symp. on Microarchitecture* (Portland, Oregon, Dec. 04 - 08, 2004). Micro '04. 355-366.

[19] Sariowan, H., Cruz R.L. G.C. Polyzos. Scheduling for quality of service guarantees via service curves. In *Proc. of the 4th Intl. Conf. on Comp. Comm. and Networks* (September 20 - 23, 1995). ICCCN '95. 512.

[20] Sha, L., Rajkumar, R., and Lehoczky, J. P. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. on Computers* 39, 9 (Sep. 1990),

[21] Sonics MemMax 2.0 Multi-threaded DRAM Access Scheduler: DataSheet, June 2006.

[22] Verghese, B., Gupta, A., and Rosenblum, M. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *Proc. of the 8th Intl. Conf. on Arch. Support For Prog. Lang. and Op. Sys.* (San Jose, CA, Oct. 02 - 07, 1998). ASPLOS-VIII. 181-192.

[23] Zhang H. Service Disciplines for Guaranteed Performance Service in Packet-switching Networks, In *Proc. of the IEEE*, vol.83, (Oct. 1995).

[24] Zhu, Z. and Zhang, Z. A Performance Comparison of DRAM Memory System Optimizations for SMT Processors. In *Proc. of the 11th Intl Symp. on High-Perf. Comp. Arch.* (Feb. 12 - 16, 2005). HPCA '05. 213-224.