Effective Management of DRAM Bandwidth in Multicore Processors

Nauman Rafique, Won-Taek Lim, Mithuna Thottethodi
School of Electrical and Computer Engineering, Purdue University
West Lafayette, IN 47907-2035
{nrafique,wlim,mithuna}@purdue.edu

Abstract

Technology trends are leading to increasing number of cores on chip. All these cores inherently share the DRAM bandwidth. The on-chip cache resources are limited and in many situations, cannot hold the working set of the threads running on all these cores. This situation makes DRAM bandwidth a critical shared resource. Existing DRAM bandwidth management schemes provide support for enforcing bandwidth shares but have problems like starvation, complexity, and unpredictable DRAM access latency.

In this paper, we propose a DRAM bandwidth management scheme with two key features. First, the scheme avoids unexpected long latencies or starvation of memory requests. It also allows OS to select the right combination of performance and strength of bandwidth share enforcement. Second, it provides a feedback-driven policy that adaptively tunes the bandwidth shares to achieve desired average latencies for memory accesses. This feature is useful under high contention and can be used to provide performance level support for critical applications or to support service level agreements for enterprise computing data centers.

1. Introduction

The two technology trends of increasing transistors per die and limited power budgets have driven every major processor vendor to multicore architectures. The number of on-chip cores is expected to increase over the next few years [1]. This increasing number of cores (and consequently threads) share the off-chip DRAM bandwidth which is bound by technology constraints [2, 4]. While on-chip caches help to some extent, they cannot possibly hold the working set of all the threads running on these on-chip cores. Thus DRAM bandwidth is a critical shared resource which, if unfairly allocated, can cause poor performance or extended periods of starvation. Because it is unacceptable that applications suffer from unpredictable performance due to the concurrent execution of other applications, DRAM bandwidth allocation primitives that offer isolated and predictable DRAM throughput and latency are desirable design goals. This paper develops simple, efficient and fair mechanisms that achieve these two goals.

The challenges in achieving these goals are twofold. First, the time to service a DRAM request depends on the state of the DRAM device which in turn depends on previous DRAM operations. This makes it hard to leverage well-understood fair queuing mechanisms since memory latency is not only variable, it also depends on the previously scheduled DRAM accesses. Second, even in the presence of fair queuing mechanisms that preserve relative bandwidth shares of contending sharers, there can still be significant variation in access latency because of (a) variation in application demand and (b) variation in DRAM state due to interleaved requests from other sharers. Allocating bandwidth shares to achieve the necessary latency for the worst case is wasteful as also observed by others [27].

Our paper develops two key innovations to address the above two challenges of managing shared DRAM bandwidth in multicore systems. **First**, we develop a fair bandwidth sharing mechanism that abstracts away the internal details of DRAM operation such as precharge, row activation and column activation and treats every DRAM access as a constant, indivisible unit of work. Our view of memory accesses as indivisible unit-latency operation is purely a logical construct to facilitate fair bandwidth sharing. In practice, our scheme does employ memory access scheduling optimizations that schedule individual DRAM commands to exploit page hits. One obvious benefit of our technique is that it enables fair sharing with lower complexity compared to a previous technique that incorporates detailed DRAM access timing [27]. One may think that this reduction in complexity is achieved at the cost of scheduling efficiency because the abstraction hides the true nature of DRAM access which is inherently variable and dependent on the state of the DRAM device. On the contrary, our technique yields improvements in the DRAM latencies observed by sharers and consequently, in overall performance. This is primarily because the simple abstraction of memory requests as an indivisible constant unit of work enables us to adopt existing fair queuing schemes with known theoretical properties. In particular, we adopt start-time fair queuing [8] which offers provably tight bounds on the latency observed by any sharer.

The above bandwidth sharing scheme can efficiently enforce specified bandwidth shares. However, the total memory latency observed by memory requests is highly variable because it depends not only on the bandwidth share but also on contention for the device. Under high contention, the latencies observed by request in memory system can be hundred of cycles more than the latencies observed without contention,

as shown by our results in Section 5.2. Our **second** innovation is a feedback-based adaptive bandwidth sharing policy in which we periodically tune the bandwidth assigned to the sharers in order to achieve specified DRAM latencies. Adaptive bandwidth management does not add to the complexity of the hardware because it can be done entirely in software at the operating system (OS) or the hypervisor by using interfaces and mechanisms similar to those proposed for shared cache management [29]. While that interface supports various features such as thread migration and thread grouping (wherein a group of threads acts as a single resource principal), our study assumes that each thread running on a processor is a unique resource principal with its own allocated bandwidth share. As such, we use the terms threads, sharers and contenders interchangeably in the rest of this paper.

Related Work: Recently, researchers have recognized the fairness issues involved in the management of DRAM bandwidth [25, 27]. Nesbit *et al.* [27] propose a fair queuing scheme for providing fairness, but their technique can suffer from starvation as we discuss in Sections 3.1 and 6. We compare our technique to theirs, referred to as *FQ-VFTF*, in our simulation results reported in Section 5.1. Moscribroda *et al.* [25] recognized the starvation problems in the work of Nesbit *et al.* [27] and proposed a scheme which keeps track of the latencies threads would experience without contention. Our scheme avoids such hardware overhead and still prevents starvation problems.

In summary, this paper makes the following contributions:

- We demonstrate a simple and efficient mechanism to achieve fair-queuing for DRAM access that improves average sharer service latency (number of cycles a sharer waits before receiving service) by 17.1% and improves overall performance by 21.7%, compared to a previously proposed DRAM fair-queuing scheme [27]. In addition, our technique reduces worst case service latencies by upto a factor of 15.
- We demonstrate that a feedback-based adaptive policy can effectively deliver predictable memory latencies (within 12% of the target in our experiments) for a preferred sharer. Such a policy can be used to prioritize critical threads or support service level agreements in data centers.

The rest of this paper is organized as follows. Section 2 offers a brief background of fair queuing and DRAM memory system organization. Then we present the details of our technique in Section 3. We discuss our experimental methodology in Section 4 and present our results in Section 5. We describe the related work in Section 6 and conclude our paper with future possibilities in Section 7.

2. Background

This study applies the ideas from fair queuing to DRAM systems. In order to give a clear understanding of the concepts in the paper, we introduce fair queuing in this section, followed by an explanation of the working of the DRAM systems.

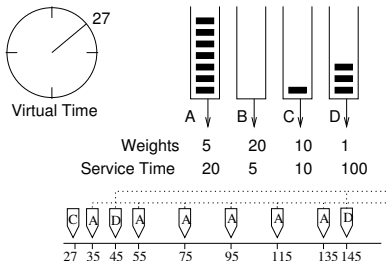


Figure 1. An example for fair queuing (adapted from [13])

2.1. Fair Queuing

The notion of fairness is well-defined and well-studied in the context of networking. Fair queuing systems are based on the concept of *max-min* fairness [5, 6]. A resource allocation is *max-min* fair if it allocates resources to sharers in proportion to their relative weights (or shares, commonly represented as ϕ). A sharer is referred to as backlogged if it has pending requests to be served. A service discipline which follows the above definition of fairness divides the available bandwidth among the backlogged queues in proportion to their share. If a sharer is not backlogged, its share of bandwidth is distributed among backlogged sharers. As soon as it becomes backlogged again, it starts receiving its share. A sharer is not given credit for the time it is not backlogged. Such credit accumulation can cause unbounded latency for the other sharers (even though these other sharers did not use the bandwidth at the expense of the first sharer). The scheduling algorithms which penalize sharers for the using idle bandwidth are generally regarded as unfair [8, 28]. Fair allocation algorithms guarantee a fair share regardless of prior usage.

Formally, an ideal fair queuing server (commonly referred to as Generalized Processor Sharing or GPS server) can be defined as follows: Suppose there are N flows and they are assigned weights of $\phi_1, \phi_2, \ldots, \phi_N$. Let $W_a(t_1, t_2)$ be the amount of flow a's traffic served during interval (t_1, t_2) during which a is continuously backlogged. Then for a GPS server,

$$\frac{W_a(t_1,t_2)}{W_b(t_1,t_2)} = \frac{\phi_a}{\phi_b}, \text{ where } b = 1, 2, \ldots, N, \text{ or}$$
$$\frac{W_a(t_1,t_2)}{\phi_a} - \frac{W_b(t_1,t_2)}{\phi_b} = 0$$

Most fair queuing implementations use a virtual clock to implement GPS (or its approximation). We explain the use of a virtual clock with an example adapted from [13]. Figure 1 shows three sharers, A, B, C and D, with weights of 5, 20, 10 and 1 respectively. At the instant shown, only A, C and D are currently backlogged. These sharers have to be served in such a way that the service received by each of them is proportional to the weights assigned to them. It is not desirable

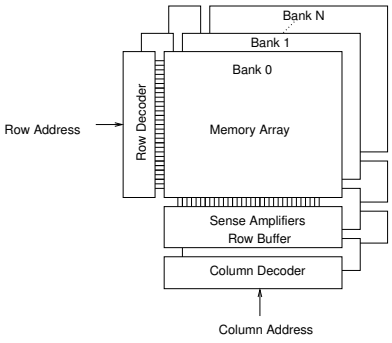


Figure 2. DRAM Device Internal Organization

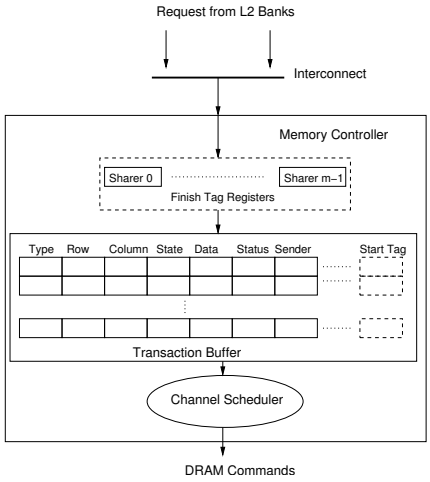


Figure 3. Architecture of Memory Controller

to visit sharers in a simple round robin manner, serving them in proportion to their weights, as that would result in requests from one sharer clustered together and others would experience long latencies [13]. We assume here that each request takes one unit of time for its service. For each sharer, we compute a service interval which is the reciprocal of its weight. Note here that weights do not have to add up to any specific value, so their value does not change when sharers become backlogged or unbacklogged. At any instant, we select the sharer which has the minimum (earliest) scheduled service time. For the instant shown in the example, sharer C has the minimum service time, i.e, 27. So the virtual time is advanced to 27 and sharer C is served. Since sharer C does not have any requests left, we do not schedule it. Next, virtual time is advanced to 35 and sharer A is served. Since A has more requests to be served, we have to reschedule it. A's service interval is 20, so its next request should be served at time 20+35=55. It is apparent that the virtual time is advanced to serve the next sharer and thus runs faster than the real time. We continue in the similar manner and the resulting service schedule is shown in Figure 1. When no sharer is backlogged, virtual time is set to zero. Once a sharer becomes backlogged, its new service time has to be computed. Fair queuing schemes differ in how they assign service time to a newly backlogged sharer.

In order to implement the above mentioned algorithm for the general case of variable sized requests, each request is assigned a start and finish tag. Let S_i^a be the start tag of i^{th} packet of sharer a and F_i^a be its finish tag, these tags are assigned as follows:

$$\begin{aligned} S_i^a &= MAX(v(A_i^a), F_{i-1}^a) \text{ and} \\ F_i^a &= S_i^a + l_i^a/\phi_a \end{aligned}$$

where $v(A_i^a)$ is the virtual arrival time of i^{th} packet of sharer a, l_i^a is the length of the packet and $F_0^a = 0$.

For an ideal (GPS) server, the virtual time is computed using server capacity and the number of backlogged queues [5, 28]. Since the number of backlogged queues change frequently (specially in DRAM system), such computation is prohibitively expensive [7, 8]. To solve this problem, Zhang *et al.* [37] proposed a virtual clock algorithm which replaces $v(A_i^a)$ with real time in the calculation of the formula for start time. A similar approach was adopted by Nesbit *et al.* [27]. This approach does not provide fairness [7, 8, 25]. We explain that with an example. Consider a bursty sharer which does not send requests for a long interval. During that interval, the virtual clocks of other backlogged sharers would make progress faster than the real clock (as the virtual clock is time scaled by the fraction of the bandwidth assigned to each sharer). When the bursty sharer finally starts sending requests, it would be assigned virtual time much smaller than the others sharers and can halt their progress for an unbounded time.

Several other fair queuing schemes have been proposed with various advantages and disadvantages. We refer the reader to [8, 36] for a survey of other approaches. For the purpose of this study, we have adopted Start Time Fair Queuing (SFQ) [8]. SFQ assigns a start and finish tag to each packet using the same formulas mentioned above. But it approximates $v(t)$ by the start time of the packet in service at time t. If no packet is in service, $v(t)$ is reset to zero. Clearly, this definition makes the computation of $v(t)$ much simpler. SFQ has strong bounds on latency, guarantees fairness even under variable rate servers, and works for arbitrary weight assignment to each sharer [8].

To apply the fair queuing techniques in the context of a DRAM system, it is important to understand DRAM operation. In the remainder of this section, we explain the internal operation of a DRAM system.

2.2. DRAM System

Modern DRAM is a multi-dimensional structure, with multiple banks, rows and columns. Figure 2 shows the internal structure of a DRAM device. Each bank of memory has an array of memory cells, which are arranged in rows and columns. For accessing data in a DRAM bank, an entire row must be loaded into the row buffer by a row activate command. Once the row is in row buffer, a column access command (read or write) can be performed by providing the column address. Now if a different row has to be accessed, a precharge command must be issued to write the activated row back into the mem-

ory array, before another row activate command can be issued. Multiple DRAM devices are accessed in parallel for a single read/write operation. This collection of devices which are operated upon together is called a rank. Multiple ranks can be placed together in a memory channel, which means that they share the same address and data bus. Each DRAM device manufacturer publishes certain timing constraints that must be respected by consecutive commands to different banks within a rank, and to the same bank. Moreover, the resource constraints have to be respected, i.e., before a column access command can be issued, it must be ensured that the data bus would be free for the column access operation. Each memory channel is managed by a memory controller, which ensures that the timing and resource constraints are met. For further details of the inner workings of DRAM devices and controllers, we refer the reader to other studies [31].

Most recent high-end processors and multicores have their memory controllers integrated on chip ([12, 15, 19]). One such memory controller is shown in Figure 3 (the part in dotted box is our addition and would be explained later). When a request arrives in the memory controller, it is placed in the transaction buffer. A channel scheduler selects a memory request to be serviced from the transaction buffer and issues the required DRAM command to the DRAM device. We would refer to a memory reference (read or write) as a memory transaction while DRAM operations would be referred to as commands. For this study, we have assumed that transaction buffer has a centralized queue instead of per bank queues. Our scheme can be easily adapted to a transaction buffer with per bank queues.

Most real memory access schedulers implement a simple First-Ready-First-Come-First-Serve mechanism for selecting DRAM commands [31]. In this mechanism, the access scheduler selects the oldest command that is ready to be issued. Rixner *et al.* [31] made the observation that this selection algorithm results in suboptimal utilization of DRAM bandwidth. For example, let us assume that DRAM system has received three memory references, all addressed to the same bank. The first and the third access the same DRAM row, while the second one accesses a different row. It would be bandwidth efficient to schedule the third transaction right after the first one, as that would save us a bank precharge and row activation operation. But this re-ordering would violate the original order in which transactions were received. Some implications of this re-ordering would be discussed in Section 3.2. Rixner *et al.* [31] studied memory access scheduling in detail and proposed that for efficient memory bandwidth utilization, memory scheduler should implement the following algorithm: (a) Prioritize ready commands over commands that are not ready (b) Prioritize column access commands over other commands (c) Prioritize commands based on their arrival order. We use this scheduling algorithm as our base case for this study and refer to it as Open-Bank-First (*OBF*) in the rest of this paper.

3. *DSFQ* Bandwidth Management Scheme

This section describes the implementation details of our scheme. We start by discussing the implementation of fair queuing in DRAM in Section 3.1. Section 3.2 discusses the tradeoff between fairness and performance. Finally, Section 3.3 describes the algorithm for our adaptive bandwidth allocation policy to achieve targeted DRAM latency for a preferred sharer.

3.1. Fair Queuing for DRAM System

In this section, we explain the modifications made to the DRAM system described in Section 2.2 to implement Start Time Fair Queuing (SFQ). DRAM systems are significantly different from simple queuing systems. Thus there are some fundamental issues to be considered before adapting any queuing discipline.

We explained in Section 2.2 that each memory transaction is divided into DRAM commands (e.g., column access, row activate, etc.) based on the state of the target DRAM bank. For a row buffer hit, only a column access command is required, while a row buffer miss requires precharge and row activate besides column access. Thus the time taken by a DRAM transaction depends on the state of bank and cannot be decided when the transaction arrives. This creates problem for calculating finish times using the formula mentioned in Section 2.1. One approach to solve this problem is to delay the calculation of finish times, until a command is issued to the DRAM device. This approach was adopted by Nesbit *et al.* [27]. They noted that by taking this approach, the commands from the same sharer might be assigned finish tags out-of-order, as memory commands are re-ordered for efficiency. Thus a younger command might receive a smaller start/finish tags than an older command. In the worst case, younger commands can keep getting issued, causing starvation or unexpectedly longer latencies for some of the older commands. The calculation of start/finish tags is further complicated by DRAM timing constraints which are dependent on the exact configuration of the DRAM system.

We propose to solve the above mentioned problems by making the implementation of fair queuing oblivious to the state and timing of DRAM device. We accomplish that by treating a DRAM transaction as a single entity (instead of composed of a variable number of commands). Memory bandwidth is allocated among sharers in units of transactions, instead of individual DRAM commands. Each transaction is considered to be a request of unit size and assumed to be serviced in a unit time. This approach simplifies the implementation of start-time fair queuing on the DRAM system. Start-time fair queuing defines the current virtual time to be the virtual start time of the packet currently in service. In our case, there are cycles in which none of the commands receives service due to DRAM timing constraints. Thus we assume the current virtual time to be the minimum start tag assigned to the currently pending transactions. This choice ensures that the worst case service latency of a newly backlogged sharer is bound by N-1 where

N is the number of sharers[1]. The finish time of the last transaction of each sharer is kept in a separate register and can be updated at the time of arrival of i^{th} transaction from sharer a as: $F_i^a = S_i^a + 1/\phi_a$. Thus if we support N sharers, we need N registers to keep track of the finish tag assigned to the last transaction of each sharer. We assume that the OS provides reciprocals of weights as integers. Thus the computation of finish tags requires simple addition. We would refer to this scheme as DRAM-Start-Time-Fair-Queuing (*DSFQ*) in the rest of the paper.

It should be noted here that the treatment of a transaction as a single entity is only for the purposes of start tag assignment. Each transaction is actually translated to multiple commands, with each command prioritized based on the start tag assigned to its corresponding transaction. One might argue that if a sharer has more row buffer conflicts than other, it might end up sending more commands to DRAM (as it needs precharge and row activate besides the column access). We assert that our scheme enforces memory reference level bandwidth shares. But it can be easily modified to keep track of command bandwidth, by updating the finish tag registers whenever a command other than column access is issued. Such a modification will still assign start tags to transactions at their arrival and thus would posses all the fairness properties of our scheme. We have experimentally tested this modification and found that it shows same results as our original scheme.

3.2. Tradeoff of Fairness and Performance

In the last section, we have provided a methodology to allocate DRAM bandwidth to sharers based on their weights. If the commands are scheduled strictly based on their start tags, the DRAM bandwidth would not be utilized efficiently. In order to utilize DRAM efficiently and support shares of each sharers, we use the following algorithm for DRAM command scheduler: (a) Prioritize ready commands over commands that are not ready. (b) Prioritize column access commands over other commands. (c) Prioritize commands based on their start tags. Note that this algorithm is similar to the one proposed by Nesbit *et al.* [27]. The only difference is that the requests are ordered based on their start tags, instead of the finish tags. There is one potential problem with the above scheduling algorithm. Consider two DRAM transactions for a same bank, one being a row hit with larger start tag and the other a row miss with smaller start tag. The above algorithm would serve the transaction with larger start tag, as it prioritizes column access commands. If there is a chain of transactions with row buffer hits but with larger start tags, they can cause unlimited delays for transactions with smaller start tags. Nesbit *et al.* [27] made a similar observation, and proposed that after a bank has been open for t_{RAS} time (timing constraint of a DRAM device – time between a row access and next precharge), it should wait for command with oldest tag to become ready.

We propose a more general and flexible solution to this problem. Whenever a DRAM command is issued, if it does not belong to a transaction with the minimum start tag, we increment a counter. If the counter reaches a specified threshold, the memory scheduler stops scheduling commands from transactions other than the one with minimum start tag and waits for those commands to become ready. Once a command from the transaction with smallest start tag is issued, the threshold counter is reset. This threshold value is called starvation prevention threshold (*SPT*). The scheduling policy mentioned above without such a threshold violates the latency bounds of SFQ. However, the introduction of *SPT* limits the increase in latency bound to a constant additive factor because at most *SPT* requests can be served before falling back to the SFQ ordering.

The optimal *SPT* value is environment- and workload-dependent. For example, if the processor is used in a situation where providing performance isolation is the key concern, a smaller threshold should be used. On the other hand, if efficiently utilizing the DRAM bandwidth is the primary goal, a higher threshold is desired. It should be noted that the efficient DRAM bandwidth utilization is not always correlated with better overall performance. In some cases, stricter enforcement of SFQ ordering between transactions (i.e., smaller thresholds) results in better performance. This is because the advantage of improved DRAM bandwidth utilization due to larger thresholds is over-shadowed by the increase in latency observed by starved transactions. Thus we propose to make this threshold adjustable by operating systems. Our results in Section 5.1 show that the optimal value of threshold varies from one benchmark mix to another and extreme (very high or very small) values of threshold hurt performance.

3.3. Adaptive Policy

The introduction of fair queuing allows us to control the relative share of each sharer in the DRAM bandwidth. But it does not provide any control over the actual latencies observed by memory references in the DRAM. The latency observed by a memory reference in DRAM depends on a lot of factors including the length of the sharer's own queue, its bank hit rate, the bank hit rate of other sharers, etc. Moreover, as the requests from different sharers are interleaved, their bank hit/miss rate would be different as some misses might be caused by interference with other sharers. Thus we cannot guarantee specific performance levels for any sharers, unless we pessimistically allocate shares for them (provisioning for worst case scenarios). Such worst case provisioning is not suitable for general purpose computing platforms [27].

We propose to solve the above mentioned problem by implementing a feedback based adaptive policy in the operating system or hypervisor. For this study, we restrict our policy to achieving target latency for only one of the sharers (extension of the policy to support target latencies for multiple sharers is part of the future work, see Karlsson *et al.* [17] for a discussion on such an algorithm). The adaptive policy tweaks the weight assigned to the target sharer (based on the feedback) to get the desired latency. The feedback can be obtained by performance

[1]The worst case occurs when every sharer has a pending transaction with the same start tag as the transaction with the smallest start tag

counters provided by processors. For this study, we assumed that performance counters for "average latency observed by requests from a processor" are available. It is expected that such counters would be made available since memory controllers are on-chip now. If such counters are not available, a feedback policy can be implemented to achieve target instructions per cycle (IPC).

(* $share_{i,t}$− *Share of sharer* i *during interval* t *)
(* $latency_{i,t}$− *Average latency of sharer* i *during interval* t *)
(* $slope_{i,t}$− *Rate of change of delay with change in share for sharer* i *during interval* t *)
(* α− *Forgetting factor* *)

Initialization:

1. `for all` i`:`
   $share_{i,0} := \frac{100}{n}$

At end of each epoch t:

1. (* *Calculate the slope for prioritized sharer* p *)
   $slope_{p,t} := \frac{latency_{p,t} - latency_{p,t-1}}{share_{p,t} - share_{p,t-1}}$
2. `if` $slope_{p,t} > 0$ `then:`
   $slope_{p,t} := slope_{p,t-1}$
3. (* *Calculate the share for the next interval* *)
   $share_{p,t+1} := share_{p,t} + \frac{(goal_p - latency_{p,t})}{slope_p}$
4. (* *Calculate EWMA of calculated share* *)
   $share_{p,t+1} := (\alpha * share_{p,t+1}) + ((\alpha - 1) * share_{p,t})$
5. (* *Divide the remaining shares among the rest of the sharers equally* *)
   `for all` $i \neq p$`:`
   $share_{i,t+1} := \frac{100 - share_{p,t+1}}{n-1}$

Figure 4. Adaptive Policy Algorithm

The algorithm used by our feedback based latency control policy is described in Figure 4. It starts out by assigning equal weights to all sharers. At the end of each epoch, it calculates "the rate of change of average latency with the change in share", called slope, during the last interval. Since the latency should decrease with the increase in share, the slope should be negative. But under certain conditions, the slope might come out to be positive. For example, consider an epoch in which the share of a sharer was increased. But during the same epoch, some new sharers become heavily backlogged. In such situations, the slope might come out to be positive. In our algorithm, we discard the positive slope and use the slope calculated previously instead. Based on the value of slope, we calculate the amount of change in share required to achieve the desired latency. In order to make sure that the transient conditions in an epoch do not cause the algorithm's response (share) to change drastically, we keep the past history of shares by calculating an exponentially weighted moving average (EWMA) of shares [10]. We use a smoothing factor α, which controls the weightage given to the past history. The value of α should be chosen in such a way that the algorithm is quick enough to respond to changing conditions, but keeps enough history to have a stable response. Our experimental results show that a value of `80` for α provides a good compromise of stability and responsiveness. After we have calculated the share of the target sharer, we divide the rest of the bandwidth equally among the rest of the sharers.

System	1 chip, 8 processors per chip
Processor Technology	65 nm, 0.5ns cycle time
Processor configuration	4-wide, 64 instruction window and 128 ROB entries, 1KB YAGS branch predictor
L1 I/D cache	128KB, 4-way, 2 cycles
L2 cache	4MB, 16-way shared, 8-way banked, 9 cycles
DRAM configuration	1GB total size, 1 channel, 2 ranks, 8 banks, 32 Transaction Buffer entries, open page policy
Disk Latency	Fixed 10ms

Table 1. Simulated configuration

Dynamically adjusting weights for the adaptive policy introduces a problem: the sharer which was assigned a small weight in the previous epoch would have its finish tag registers and start tags of pending transactions set to high values. That would mean that the sharer would be de-prioritized (at least for some time) during the next epoch too, even if its weight is increased. Incorrect start tags of pending transactions is not a significant problem for DRAM, as the DRAM transaction queue size is bounded (`32` for this study). Time required for serving the pending transactions would be a very small fraction of the total epoch time (see Zhang *et al.* [38] for similar arguments). For finish tag registers, we propose a simple solution: whenever the weights of sharers are updated, the finish tag register for each sharer is reset to the maximum of all finish tags. Thus all new memory transactions would get start tags which would be based on their new weight assignment.

For the implementation of this adaptive policy, we selected epoch length of 2 million cycles. We found the execution time of the algorithm to be `15 microseconds` on average; thus the timing overhead of the adaptive scheme is only 1.5%. Our results in Section 5.2 show that the adaptive policy successfully achieves the target latencies.

Summary: We have proposed a scheme which enforces OS specified shares, uses the DRAM bandwidth efficiently, avoids starvation and is flexible. We have also suggested a policy for adjusting weights to achieve target memory latency for a prioritized sharer. Next, we describe the experimental methodology used to evaluate our scheme.

4. Methodology

We used the Simics [21]-based GEMS [22] full system simulation platform for our simulations. The simulated system runs an unmodified Solaris operating system version 5.9

Workload	Benchmarks	Benign
Mix_1	ammp, mcf, lucas, parser, vpr, twolf	applu
Mix_2	facerec, equake, swim, gcc, mgrid, gap	apsi
Mix_3	equake, mcf, ammp, swim, lucas, applu	vpr
Mix_4	vpr, facerec, gcc, mgrid, parser, gap	fma3d
Mix_5	parser, gap, apsi, fma3d, perbmk, crafty	art

Table 2. Workloads

and simulates a 8-core CMP. Each core is modeled by OPAL module in GEMS. OPAL simulates an out-of-order processor model, implements partial SPARC v9 ISA, and is modeled after MIPS R10000. For our memory subsystem simulation, we use the RUBY module of GEMS which is configured to simulate an MOESI directory-based cache coherence protocol. We use a 4-way 128KB L1 cache. The L2 cache is 8-way banked, with a point-to-point interconnect between the L2 banks and L1. For the L2 cache, we used a modification of cache replacement scheme to support fairness as suggested by cache fairness studies [18, 29, 34]. This replacement scheme enforces per processor shares in the L2 cache (we set equal shares for all processors). This scheme ensures that the shared cache is used efficiently while still providing performance isolation between sharers. We used CACTI-3.0 [32] to model the latency of our caches and DRAMsim memory system simulator [35] to model the DRAM system. We modified DRAMsim to implement the transaction scheduling policies studied in this paper. DRAMsim was configured to simulate 667 MHz DDR2 DRAM devices, with timing parameters obtained from the Micron DRAM data-sheets [24]. We used open page policy for row buffer management (row buffer kept open after a column access), instead of closed page policy (row buffer closed after each column access), as our base case command scheduling policy exploits row buffer hits. Table 1 lists the key parameters of the simulated machine.

We used five multiprogrammed workloads for this study as shown in Table 2. These workloads include the most memory intensive applications from SPEC 2000 suite as mentioned by [39]. In order to test the effectiveness of our performance isolation policy, we use a kernel that acts as a DRAM bandwidth `hog`. The `hog` creates a matrix bigger than the cache space and accesses its elements in such a way that its each access is a miss. For the experiments with `hog`, it replaces the benchmark listed as "Benign" (Table 2) in the mixes. For our simulations, caches are warmed up for 1 billion instructions and the workloads are simulated in detail for 100 million instructions. In order to isolate the results of our benchmarks from OS effects, we reserve one (out of eight) core for the OS and run benchmarks on the remaining 7 cores. We simulate the adaptive policy inside the simulator without modifying the operating system.

5. Results

The two primary results of our experiments are:

1. Our fair queuing mechanism offer better performance isolation compared to *FQ-VFTF* proposed by Nesbit *et al.* [27]. The improvements are visible in both fairness (improvement in maximum sharer service latency) and performance (21% improvement in IPC).

2. The feedback-driven adaptive policy achieves latencies within 12% of the target. The policy shows a behavior which is stable and responsive with the smoothing factor of `80`.

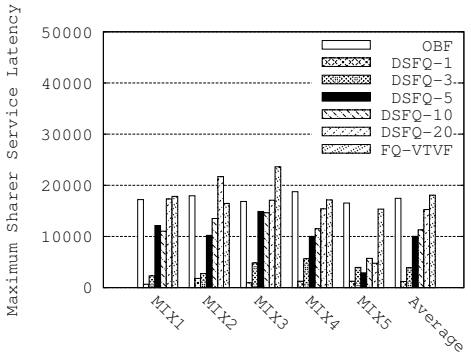


Figure 5. Worst-case Sharer Service Latencies

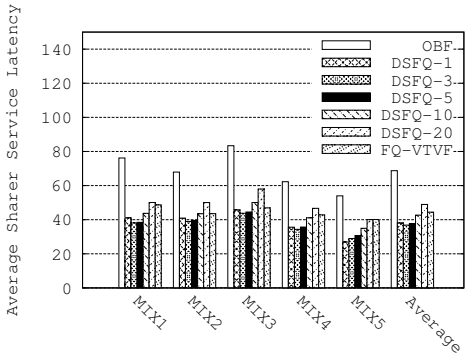


Figure 6. Average Sharer Service Latencies

We elaborate on these two results in the remainder of this section.

5.1. Performance Isolation

In this section, we will present results for experiments performed to study the performance isolation or fairness capabilities of different transaction scheduling policies. In these experiments, `hog` is included as a sharer and the share (ϕ) of each sharer is set to be equal. In order to analyze the fairness and/or starvation behavior of different transaction scheduling policies, we measured sharer service latencies for each DRAM command. Sharer service latency for i^{th} command from sharer a is calculated as follows:

$$sharer\_service\_latency_i^a = t - MAX(arrival_i^a - last\_service^a)$$

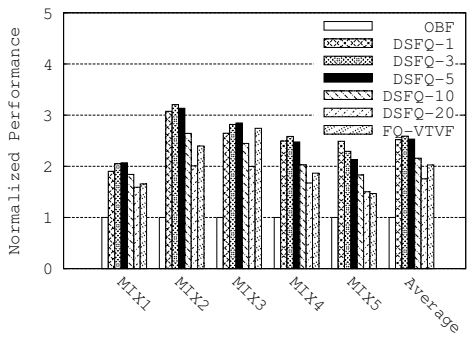


Figure 7. Normalized IPC

where t is the time that the command is issued, $arrival_i^a$ is the arrival time for the command, and the $last\_service^a$ is the time when the last command from the same sharer was issued.

The strength of a fair queuing algorithms is often represented using the worst case latencies observed by sharers under that policy [3], as it highlights the possibilities of starvation. We adopt the same approach and present the maximum (worst case) sharer service latencies of different transaction scheduling policies for all benchmarks in Figure 5. It can be observed that the *OBF* policy has extremely high sharer service latencies. In Section 3.2, we introduced *SPT* as a measure to avoid starvation and expect small values of *SPT* to avoid starvation. The results in Figure 5 clearly show that behavior. On average, the worst case service latencies increase with an increase in *SPT* value. One can notice that for some mixes (Mix_1, Mix_3 and Mix_5), the relationship between worst case service latency and *SPT* is not monotonic, i.e., a higher *SPT* has a smaller worst case latency. The worst case latency depends on the dynamics of the system and the worst case is not guaranteed to happen always. It can be observed that *FQ-VFTF* has high worst case sharer service latencies, in some cases even higher than *OBF* and 15 times higher than $DSFQ_3$ on average. There are multiple reasons behind this behavior (they were mentioned earlier in Section 2.1 and in Section 3.1). First, *FQ-VFTF* assigns finish tags to commands of a sharer out-of-order, and second, a bursty sharer can cause longer latencies for other sharers once it becomes backlogged.

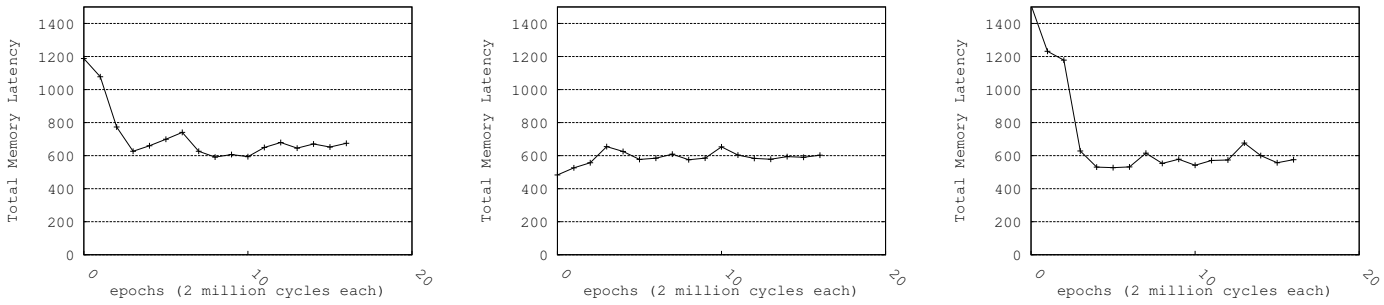
It should be noted here that the worst case behavior is not representative of performance, as it might occur only rarely. For performance, average or common case behavior is considered more relevant. Thus we shift our focus to average sharer service latencies. Figure 6 plots average sharer service latencies for different transaction scheduling mechanisms. *OBF* results in the worst latency of all mechanisms (`68.7` cycles) on average, while $DSFQ_3$ shows best latency (`36.8` cycles). It should be noted here that the average sharer latency of $DSFQ_1$ is slightly higher than $DSFQ_3$. The reason is that $DSFQ_1$ strictly enforces DRAM command ordering, and does not utilize the DRAM system very efficiently. *FQ-VFTF* has average sharer service latency of `44.4` cycles on average, which is `17.1%` worse than $DSFQ_3$. It should also be noted that *FQ-VFTF* has better sharer service latencies on average than $DSFQ_{20}$ (which was not true for worst case sharer service latencies). This shows that *FQ-VFTF* has better average case behavior than its worst case behavior.

Figure 7 shows the normalized IPC of all benchmarks with different transaction scheduling mechanisms. It can be observed that performance trends are similar to the average sharer service latency. $DSFQ_3$ gets the best performance on average, followed by $DSFQ_1$ and $DSFQ_5$. $DSFQ_3$ improves performance by `158%` over *OBF* and by `21.7%` over *FQ-VFTF*. *FQ-VFTF* shows better performance than $DSFQ_{20}$ and improves performance by `102%` over *OBF*. It should be noted that the best performing policy is different for different mixes. For Mix_1 and Mix_3, $DSFQ_5$ performs the best. In case of Mix_2 and Mix_4, $DSFQ_3$ has the best performance, while $DSFQ_1$ works best for Mix_5. That confirms the observations we made in Section 3.2. In some cases, serving request in the exact order dictated by start tags can hurt performance, as it would increase the number of bank conflicts (as shown by Mix_1, Mix_2, Mix_3 and Mix_4 for $DSFQ_1$). In other cases, prioritizing ready commands and row buffer hits can hurt the performance as the latency of other requests increases, as can be seen in Mix_5. In general, we observed that *SPT* values in the 1 to 5 range are ideal choices if *SPT* is to be statically chosen. One may also consider a feedback-based adaptive approach (similar to the one described in Section 3.3) for automatically tuning the *SPT* for any application mix. We do not study adaptive *SPT* tuning in this paper.

5.2. Adaptive Policy

In this section, we will present results for the adaptive policy explained in Section 3.3. For these experiments, we replace `hog` by the benchmarks listed as *Benign* in Table 2. We restrict ourselves to Mix_1, Mix_2 and Mix_3 for these experiments, as the other two mixes do not have a significant level of contention in memory. The target latency is set to be `600` cycles for these experiments. We selected one benchmark from each mix (`lucas`, `swim` and `ammp` for Mix_1, Mix_2 and Mix_3 respectively) as the prioritized application. In our simulations without adaptive policy, we noticed that `lucas`, `swim` and `ammp` had average memory latencies of `1267`, `504` and `1273` cycles in their respective mixes. Thus the target latencies are lower than the observed latencies for `lucas` and `ammp` and higher for `swim`. We made this choice to study the capability of the adaptive algorithm for both kind of targets. A target higher than actual latency can be useful, if the targeted sharer can be de-prioritized to improve the latencies of other contenders. We chose $DSFQ_5$ as the underlying transaction scheduling mechanism for these experiments. We performed our experiments with different values of smoothing factor (α in Figure 4) and found that the smoothing factor of `80` provides a good compromise between agility and stability of the algorithm.

Figure 8 plots the memory latencies achieved by the adaptive policy for the three selected benchmarks. It can be observed that the algorithm reaches the target latencies in 3



(a) Mix1 (b) Mix2 (c) Mix3

Figure 8. Latency Achieved by Adaptive Scheme

epochs for all the three mixes. For Mix_2 and Mix_3, there is a small overshoot. In the case of Mix_1, the latency starts to increase after reaching the target (due to increased number of DRAM requests) but the algorithm detects that change and adjusts the share accordingly. The algorithms sets the shares of `lucas`, `swim`, and `ammp` to 35%, 9% and 37% respectively in the steady state (share numbers are not shown in interest of space). The algorithm achieves average latencies within 12%, 2.5% and 9% of the target latency for the three benchmarks. That results in an IPC improvement of 64% and 38.5% for `lucas` and `ammp` respectively; and a slight reduction in IPC of `swim` with a little increase in the performance of other benchmarks in Mix_2.

The above results show that the adaptive algorithm is stable, responsive and achieves the target latencies. It can also be observed that such an adaptive algorithm can be used for effective prioritization, as it improves the IPC by up to 64% for one of the benchmarks. We also performed some experiments in which the latency target was too low to be achieved. In such cases, the adaptive policy settled at the closest achievable latency to target.

6. Related Work

Memory bandwidth management for chip multiprocessors have recently attracted attention of researchers. Nesbit *et al.* [27] proposed a scheme for providing QoS in the memory controller of a CMP. They implemented fair queuing, but they used real time, instead of virtual time in their calculation of finish tags. They base this choice on the argument that a thread which has not been backlogged in the past should be given higher share if it becomes backlogged. We noticed that if such a thread has a bursty behavior, it can cause starvation for other threads (as shown in our results in Section 5.1). Hence we make the case that if a thread does not use excess bandwidth at the cost of other threads, it should not be penalized. This notion has been widely accepted in networking community [8]. Moreover, in their scheme, finish tag is assigned to each command and is calculated using the actual time taken by the command (which is not known until a command is issued). Thus in their scheme, memory references are not assigned finish tags in their arrival order. We make the observation that this can cause starvation and unexpectedly long latencies for some memory references (See Section 3.1 for further details). Nesbit *et al.*suggest that their scheme provides QoS but concede that it cannot guarantee any limits on the overall latency observed by memory references. Our scheme not only solves the above mentioned problems, but also simplifies the implementation of the hardware mechanism.

Iyer *et al.* [14] used a simple scheme for DRAM bandwidth allocation, and refereed to the work by Nesbit *et al.* [27] for more sophisticated DRAM bandwidth management mechanism. Moscibroda *et al.* [25] studied the fairness issues in DRAM access scheduling. They defined the fairness goal to be "equalizing the relative slowdown when threads are co-scheduled, relative to when the threads are run by themselves". Thus their scheme has additional overhead for keeping track of row buffer locality of individual threads. Natarajan *et al.* [26] studied the impact of memory controller configuration on the latency-bandwidth curve. Zhu *et al.* [39] proposed the memory system optimizations for SMT processors. There have been many proposals discussing memory access ordering [11, 23, 30, 31]. We have used an aggressive memory access ordering scheme as our base case in this study.

Some memory controller proposals provide methods of supporting hard real time guarantees on memory latencies [9, 20, 33]. These schedulers either rely on knowing the memory access pattern and are not suitable for general purpose systems or compromise efficiency for providing QoS. Our scheme is flexible as it lets the OS choose the right balance between efficiency and QoS. Moreover, it is based on feedback based control and do not make any assumptions about arrival pattern of memory requests. Feedback based control of weights in fair queuing is studied in other contexts [16, 38]. Our adaptive scheme is designed using the observations made by these studies. (The design of a basic feedback-based control algorithm is not claimed as our contribution).

7. Conclusion

In this paper, we proposed a scheme for managing the DRAM bandwidth. Our scheme is simple, fair and efficient.

Simplicity is achieved by treating each memory request as a unit of scheduling (instead of composed of multiple DRAM commands). The simplicity of our scheme enables *fairness* by allowing us to use start time fair queuing with proven bounds on worst case latency. The scheme utilizes DRAM *efficiently* as it schedules commands to maximize row buffer hits, unless such scheduling order starts conflicting with fairness goals. Enforcing DRAM bandwidth shares is not adequate to achieve predictable memory access latencies. Thus we introduced an adaptive algorithm which adjusts the weight assignment of sharers to achieve desired latencies.

Our results show that our scheme provides very good average and worst case sharer service latencies. It shows a performance improvement of 21% on average over *FQ-VFTF*. Our adaptive policy achieves latencies within 12% of the target, and results in performance improvement of up to 64%. In future, we plan to explore more sophisticated adaptive policies, which try to achieve target latencies for multiple sharers.

Acknowledgments

The authors would like to thank Asad Khan Awan for discussions on network fair queuing. We would also like to thank anonymous reviewers for their feedback. This work is supported in part by National Science Foundation (Grant no. CCF-0644183), Purdue Research Foundation XR Grant No. 690 1285-4010 and Purdue University.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.

[2] S. I. Association. ITRS Roadmap, 2005.

[3] J. C. R. Bennett and H. Zhang. WF 2 q: Worst-case fair weighted fair queueing. In *INFOCOM (1)*, pages 120–128, 1996.

[4] D. Burger, J. R. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd annual international symposium on Computer architecture*, pages 78–89, New York, NY, USA, 1996. ACM Press.

[5] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*, pages 1–12, New York, NY, USA, 1989. ACM Press.

[6] E. M. Gafni and D. P. Bertsekas. Dynamic Control of Session Input Rates in Communication Networks. *IEEE Transactions on Automatic Control*, 29(11):1009–1016, November 1984.

[7] S. J. Golestani. A Self-Clocked Fair Queueing Scheme for Broadband Applications. In *INFOCOM*, pages 636–646, 1994.

[8] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Trans. Netw.*, 5(5):690–704, 1997.

[9] H. P. Hofstee, R. Nair, and J.-D. Wellman. Token based DMA. United States Patent 6820142, November 2004.

[10] J. S. Hunter. The Exponentially Weighted Moving Average. *Journal of Quality Technology*, 18(4):203–210, 1986.

[11] I. Hur and C. Lin. Adaptive History-Based Memory Schedulers. In *Proceedings of the 37th annual International Symposium on Microarchitecture*, pages 343–354, 2004.

[12] S. M. Inc. Ultrasparc IV processor architecture overview, Februrary 2004.

[13] A. Ioannou and M. Katevenis. Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High Speed Networks. *IEEE/ACM Transactions on Networking*, 2007.

[14] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. *SIGMETRICS Perform. Eval. Rev.*, 35(1):25–36, 2007.

[15] R. N. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, 2004.

[16] M. Karlsson, C. Karamanolis, and J. Chase. Controllable fair queuing for meeting performance goals. *Performance Evaluation*, 62(1-4):278–294, 2005.

[17] M. Karlsson, X. Zhu, and C. Karamanolis. An adaptive optimal controller for non-intrusive performance differentiation in computing services. In *International Conference on Control and Automation (ICCA)*, pages 709–714, 2005.

[18] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *IEEE PACT*, pages 111–122, 2004.

[19] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005.

[20] K.-B. Lee, T.-C. Lin, and C.-W. Jen. An efficient quality-aware memory controller for multimedia platform soc. *IEEE Transactions on Circuits and Systems for Video Technology*, 15(5):620–633, May 2005.

[21] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[22] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 2005.

[23] S. A. McKee, A. Aluwihare, B. H. Clark, R. H. Klenke, T. C. Landon, C. W. Oliver, M. H. Salinas, A. E. Szymkowiak, K. L. Wright, W. A. Wulf, and J. H. Aylor. Design and Evaluation of Dynamic Access Ordering Hardware. In *International Conference on Supercomputing*, pages 125–132, 1996.

[24] Micron. DDR2 SDRAM Datasheet.

[25] T. Moscibroda and O. Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. Technical Report MSR-TR-2007-15, Microsoft Research, February 2007.

[26] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, pages 80–87, New York, NY, USA, 2004. ACM Press.

[27] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *MICRO '06: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 208–222, Washington, DC, USA, 2006. IEEE Computer Society.

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

[29] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural Support for Operating System-Driven CMP Cache Management. In *The Fifteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2006.

[30] S. Rixner. Memory Controller Optimizations for Web Servers. In *Proceedings of the 37th annual International Symposium on Microarchitecture*, pages 355–366, 2004.

[31] S. Rixner, W. J. Dally, U. J. Kapasi, P. R. Mattson, and J. D. Owens. Memory access scheduling. In *International Symposium on Computer Architecture*, pages 128–138, 2000.

[32] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power and Area Model. Technical report, Western Research Lab - Compaq, 2001.

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

[34] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[35] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMsim: A memory-system simulator. *SIGARCH Computer Architecture News*, 33(4):100–107, September 2005.

[36] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks, 1995.

[37] L. Zhang. Virtual clock: a new traffic control algorithm for packet switching networks. In *SIGCOMM '90: Proceedings of the ACM symposium on Communications architectures & protocols*, pages 19–29, New York, NY, USA, 1990. ACM Press.

[38] R. Zhang, S. Parekh, Y. Diao, M. Surendra, T. F. Abdelzaher, and J. A. Stankovic. Control of fair queueing: modeling, implementation, and experiences. In *9th IFIP/IEEE International Symposium on Integrated Network Management*, pages 177–190, May 2005.

[39] Z. Zhu and Z. Zhang. A Performance Comparison of DRAM Memory System Optimizations for SMT Processors. In *Proceedings of 11th International Sysmposium on High Performance Computer Architecture*, pages 213–224, 2005.