Scalable QoS-Aware Memory Controller for High-Bandwidth Packet Memory

Hyuk-Jun Lee, *Member, IEEE*, and Eui-Young Chung, *Member, IEEE*

***Abstract*—This paper proposes a high-performance scalable quality-of-service (QoS)-aware memory controller for the packet memory where packet data are stored in network routers. A major challenge in the packet memory controller design is to make the design scalable. As the input and output bandwidth requirement and the number of output queues for routers increase, the memory system becomes a bottleneck that limits the performance and scalability. Existing schemes require an input and output buffer that store packet data temporarily before they are written into or read from the memory. With the buffer size proportional to the number of output queues, the buffer becomes a limiting factor for scalability. Our scheme consists of a hashing logic and a reorder buffer whose size is not proportional to the number of output queues and is scalable with the increasing number of output queues. Another major challenge in the packet memory controller design is supporting QoS. As an increasing number of internet packets become latency sensitive, it is critical that the memory controller is capable of providing different QoS to packets belonging to different classes. To the best of our knowledge, no published work on the packet memory controller supports QoS. In this paper, we show our scheme reduces the SRAM buffer size of the existing schemes by an order of magnitude whereas guaranteeing a packet loss probability as low as 10^{-20}. Our QoS-aware scheduler shows that it meets the latency requirements assigned to multiple service classes under dynamically changing input loads for multiple classes using a feedback control loop.**

***Index Terms*—High-performance memory system, memory controller, packet memory.**

I. Introduction

NETWORK routers store and forward high-speed Internet protocol (IP) packets. The bandwidth of the transmission lines for the routers, often called line-rate, increases rapidly with the increasing bandwidth requirement and advance in the optical technology [1], [2]. The line-rate increases from 40 Gb/s [optical carrier (OC)[1]-768] to 160 Gb/s (OC-3072) and beyond, and twice the line-rate is required for the memory bandwidth to store and retrieve data into and from the memory of the routers. A packet is broken into smaller fixed size data called *cells* and cells are written into and read from the memory. DRAMs are widely used to build a packet memory. To compensate for slow DRAM access, hundreds or thousands of pins are often used to access lots of bits in parallel.

To close the gap between slow DRAM and high-speed core logic, many schemes were proposed in both computer architecture and network areas. Some works [9], [10] try to improve the performance of DRAM components. Other works [6]–[8] try to reduce the bank conflicts in order to improve the average bandwidth utilization. However, these schemes are not free from packet loss due to bank conflicts.

Iyer *et al.* [3] proposed a scheme which guarantees no packet loss at the cost of SRAM buffer. The SRAM buffer consists of input and output buffers that are maintained per output queue. In this scheme, cells are stored temporarily in the SRAM buffer before being written into or after being read from the memory. This SRAM buffer size is proportional to the number of output queues in the system and the DRAM access time. A rapidly increasing number of output queues in modern routers make this scheme very costly in terms of die area. García *et al.* improved this scheme by overlapping multiple accesses and random bank selection scheme [4], [5]. Among all previous works, [4] is only one that is reported to be reasonably scaled up to OC-3072 or higher. However, this scheme also suffers from a large area penalty since the SRAM size is still proportional to the number of output queues.

Another major issue in the packet memory controller design is providing different quality-of-services (QoSs) to the packets belonging to different service classes. Internet traffic is often classified into multiple service classes. According to DiffServ definitions [17], different service classes of internet traffic require different latency and packet loss requirements. To satisfy these requirements, packets inside routers are classified and handled according to their service classes. This becomes increasingly important since voice over Internet Protocol (VoIP) and video packets become a significant portion of internet traffic and they are latency sensitive. Providing QoS is widely discussed in many different applications. Relatively closely related works include [11]–[16]. These works use the feedback control theory to provide QoS to web servers. Their approaches are software techniques to control either service rate or response latency of the web servers according to the classification of packets. To the best of our knowledge, no published work on the packet memory controller provides QoS.

Our proposed method consists of a hashing and a reorder buffer whose size is not proportional to the number of output queues and is scaled up to OC-3072 and higher. The reorder buffer consists of bank FIFOs and arbiters/schedulers. Our proposed QoS-aware scheduler is capable of providing QoS to the packets of different classes using a feedback control loop.

Manuscript received November 17, 2006; revised April 10, 2007. This work was supported in part by the Basic Research Program of the Korea Science and Engineering Foundation under Grant R01-2006- 000-10156-0 and by Korean Ministry of Information and Communications under the IT R&D Project.

H.-J. Lee is with the Cisco Systems, San Jose, CA 95134 USA (e-mail: hyukjunl@cisco.com).

E.-Y. Chung is with the School of Electrical and Electronic Engineering, Yonsei University, Seoul 120-749, Korea (e-mail: eychung@yonsei.ac.kr).

Digital Object Identifier 10.1109/TVLSI.2007.915367

[1]OC levels describe a range of digital signals that can be carried on SONET fiber optic network. The data rate for OC-n is $n \times 51.8$ Mbits/s.

The feedback control loop dynamically changes the scheduling ratios among different classes as the input loads for different classes change over time and meets the latency requirements assigned to different classes. Our study shows that our scheme virtually does not suffer from packet loss and leads to a much less SRAM size and access latency while meeting the QoS requirements.

Section II describes the packet memory system which includes the baseline router architecture, the memory architecture, and the scalable memory controller. Section III describes the simulator architecture and energy model. Section IV presents the simulation results. Section V discusses the packet loss probability and the cost of our scheme by comparing the results with existing schemes. Section VI gives our conclusion.

II. Packet Memory System

A. Baseline Router Architecture

The basic function of a router is to receive IP packets through its input ports, find the output port based on IP address table lookup, and forward packets to the output ports. Packets are classified inside routers based on their service classes and broken into fixed size cells. The cells are stored into output queues before being scheduled and sent to the output ports. Output queues may represent different service classes or different flows such as user datagram protocol (UDP) or transmission control protocol (TCP) connections, video streaming, or VoIP. Output queues are used to provide different levels of buffering or different QoS to different traffic streams. Multiple output queues are grouped and mapped onto an output port (or a line interface, e.g., OC192). Thus, as line-rate increases, routers should support more output queues.

The output queues are built in the packet memory. The packet memory is made of DRAMs [4]. Due to the speed mismatch between fast core logic and slow DRAMs, bank interleaving is used. To handle the bank conflicts, the packet memory controller is equipped with read and write first-inputs–first-outputs (FIFOs) which store cell read and write requests temporarily. These FIFOs are often implemented as on-chip SRAMs for high-speed access. Upon a write into the packet memory, a cell is written into the write FIFO before being sent to the DRAM. Upon a read, a cell read request is stored into the read FIFO before being sent to the DRAM. DRAM returns the requested cell into the read data buffer. Further information on the router architecture can be found in [4] and [22].

B. Memory Architecture

The packet memory system is built from multiple DRAM parts to provide a sufficient storage space to hold cells during the round trip time (RTT). Several works discuss the optimal buffering in core routers to avoid packet loss [4], [19]–[21]. In [4], it was argued that roughly one gigabytes of memory is required for OC-768 to avoid packet loss upon congestion. Fig. 1 shows a logical view of the packet memory. The packet memory stores cells. Cells are connected through linked lists. The linked lists are maintained per output queue as shown in Fig. 1. If the cell size is C bytes and the packet memory size is N bytes, the packet memory stores N/C cells. To address this cell, we need $log_2(N/C)$ bits. To maintain linked lists, we need $(N/C) \times log_2(N/C)$ bits for a pointer array. Previous works [3]–[5] used 64 bytes for the cell size. We used the same cell size for a fair comparison. If $C = 64$ bytes, $N = 1$ GB, the packet memory stores a total of 16 million cells and a pointer array is 16 M $\times$ 24 bits. This is too large to be implemented on a die. For this reason, the memory is partitioned into bigger memory blocks that can hold multiple cells. And the linked lists for these blocks are built instead as shown in Fig. 1. In the example shown in Fig. 1, a block contains eight cells and a total of 2 M blocks constitutes the packet memory. Within the block, consecutive cells are addressed sequentially. One head and tail pointer per output queue are maintained to access the beginning and end of the linked list upon reading and writing a cell. A cell in the packet memory is addressed using a block address and a block offset. In Fig. 1, the first cell in the output queue n is pointed by the head pointer and its offset and the last cell is pointed by the tail pointer and its offset. A new block is allocated when a cell comes in and the last block of the linked list for the output queue does not have space to store a cell. A block is deallocated when all cells stored in the block are read.

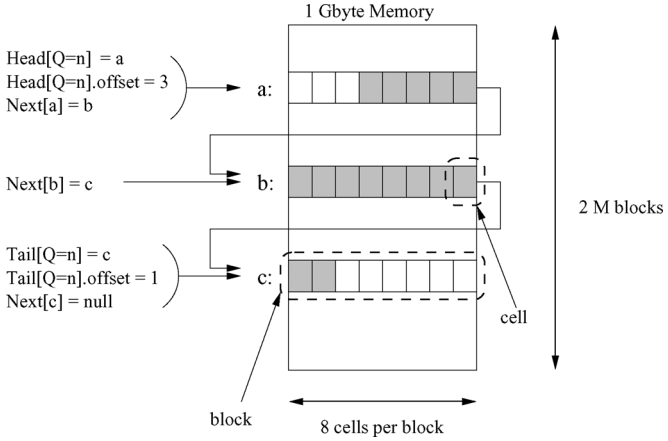


Fig. 1. Logical packet memory view: blocks containing consecutive cells are connected by linked lists. The first block and the last block are pointed by the head and tail pointer for output $\mathrm{queue} = \mathrm{n}$ separately.

A modern DRAM part consists of multiple banks to hide long access latency and adopts a burst read and write to access a large number of bits at once. To get the best performance out of the memory system with a given number of parts and banks per part, it is crucial to find an optimal cell mapping on multiple parts and banks.

Fig. 2 shows timing diagrams for the cell writes in two different cell mappings. In this example, the memory system consists of four DRAM parts and four banks per part. In Fig. 2, four cells are about to be written into the packet memory and the first three cells go to the output queue 0 and the next cell goes to the output queue 1. In both Fig. 2(a) and (b), four cells are written into four DRAM parts. In Fig. 2(a), the first cell is written into four banks (B0, B1, B2, B3) of the part 0 (P0) over four DRAM bursts. The second, third, and fourth cell are written into P1, P2, and P3, respectively, in parallel with the first cell. In Fig. 2(b), the first cell is written into the bank 0 of the four parts at the first DRAM burst. The second, third, and fourth cell are written

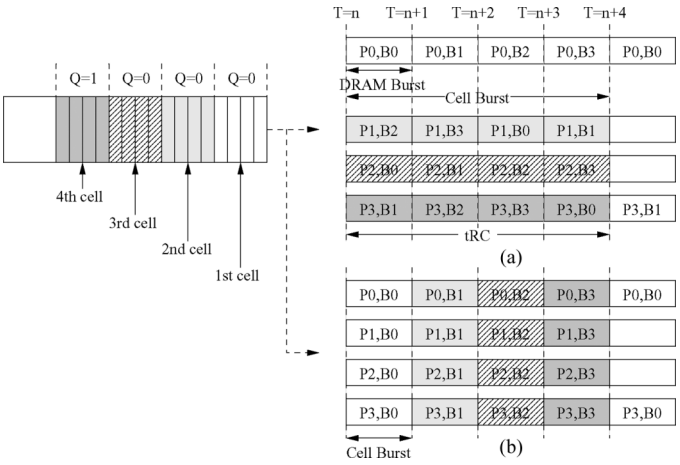


Fig. 2. Timing diagram for writing a cell in two cell mappings. (a) Cell is mapped on four banks in one DRAM part. (b) Cell is mapped on four banks across four DRAM parts. Four different colors represent four cells, respectively.

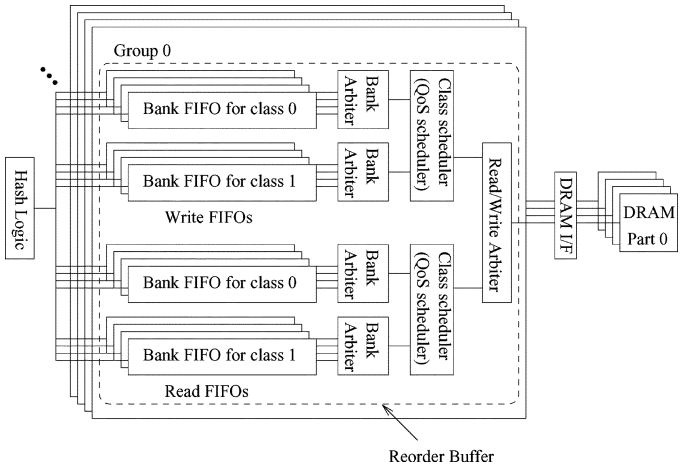


Fig. 3. SQMC for the memory system that consists of four groups and two classes (four banks per group or class).

into B1, B2, and B3 of four parts during next three consecutive DRAM bursts. We define a *cell burst* as the number of DRAM bursts used to access a cell. In Fig. 2(a), the cell burst is four DRAM bursts whereas it is one for Fig. 2(b).

To better represent various mappings, we define two terms: group and logical bank. A *group* is defined as a collection of DRAM parts that need to be accessed for writing or reading a single cell. A *logical bank* is a collection of banks where a single cell is distributed over. Fig. 2(a) has four groups and each group has one logical bank, whereas Fig. 2(b) has one group and the group has four logical banks.

A three-tuple, (G, B, C), is used to represent a cell mapping. G, B, and C represent the number of groups, the number of logical banks per group, and the cell burst size, respectively. The three-tuple for Fig. 2(a) and Fig. 2(b) are (4, 1, 4) and (1, 4, 1). In the following sections, we refer to a logical bank when we say "bank."

C. Scalable QoS-Aware Memory Controller

Two building blocks of the scalable QoS-aware memory controller (SQMC) are a hashing logic and a reorder buffer as shown in Fig. 3. The hashing logic takes an address for the cell write or read request as an input and remaps the address into another address so that the consecutive cell writes or reads are distributed over multiple memory groups and banks as evenly as possible. The write or read address consists of a block address and a block offset. The hash function takes a block address and a block offset as inputs and produces a group number, a bank number, and a bank address as outputs. The hash logic will be further explained in Section II-C1.

Even with a perfect hashing function, it is not possible to avoid bank conflicts. A reorder buffer is used to buffer memory read or write requests upon bank conflicts in computer applications [7]. The reorder buffer reorders cell requests when the bank conflict happens. That is, the requests are not dequeued from the bank FIFOs in the order of enqueues. The later requests can be sent to the DRAMs before the earlier requests by the schedulers depending on the chosen scheduling scheme. The reorder buffer can be implemented in two different ways: a shared bank FIFO or a per-bank FIFO. In the shared bank FIFO scheme, bank FIFOs for all logical banks are combined into one shared FIFO whereas in the per-bank FIFO scheme, the bank FIFO for each logical bank is implemented separately. The shared bank FIFO is usually more complex to implement while it takes less area. Fig. 3 shows the per-bank FIFO implementation. In this paper, we use per-bank FIFO. The per-bank FIFO will be called bank FIFO throughout the paper.

Another major component of reorder buffer is three levels of arbiters and schedulers: bank arbiter, class scheduler (or QoS-aware scheduler), and read/write arbiter. The bank arbiter chooses one schedulable (not busy) bank based on the arbitration method. This will be further discussed in Section II-C2. The class scheduler chooses a class based on the weighted round robin algorithm. This will be further discussed in Section II-C-3. The read/write arbiter is just a simple round-robin arbiter which alternates read and write scheduling. We have separate read and write FIFOs because a write request consists of the data payload along with a write address whereas a read request consists of only a read address.

1) Hash Logic: Assuming that a block address, a block offset, a group number, and a bank number are i, j, m, and n bits, respectively, the hashing function is defined in (1)–(5). And its implementation is shown in Fig. 4

$$cell\_pos = (block\_offset[j-1:0] + block\_addr[j-1:0])\,\%2^j \quad (1)$$

$$mem\_addr = ((block\_addr[i-1:0] \ll j)\,|cell\_pos) \quad (2)$$

$$group = mem\_addr[m-1:0] \quad (3)$$

$$bank = mem\_addr[n+m-1:m] \quad (4)$$

$$bank\_addr = mem\_addr[i+j-1:n+m]. \quad (5)$$

In (1) and (2), %, $\ll$, and | represent modulo, bit left shift, and bitwise OR, respectively. In (2), the bitwise OR is same as concatenation because $cell\_pos$ is j-bit wide. $cell\_pos$, $block\_addr$, mem_addr, and $bank\_addr$ in (1)–(5) represent cell position,

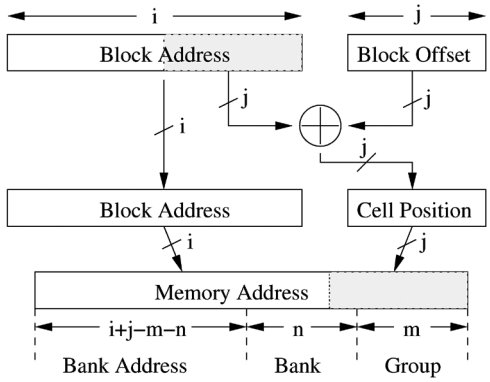


Fig. 4. Implementation of the hashing function.

block address, memory address, and bank address, respectively. They are explained in detail in the next paragraph.

In the proposed scheme, when a cell comes in and there is no space to store the cell, i.e., an output queue is empty of cells or the last block is full of cells, a free block is allocated randomly. The block address is randomly chosen because we use its lower j bits to determine the group and bank number and the random address gives a better hashing performance. In our scheme, the cell position within a block is computed by (1). That is, the cell position is shifted by the lower j bits of the block address. Without this shifting, writing into a block always starts with the block offset 0. The shifting randomizes the initial block offset and thus reduces potential conflicts that may be possible when many blocks are synchronously allocated and writings to them start from the block offset 0. This cell position is concatenated with a block address in (2) to produce a memory address. The lower m bits of the memory address are assigned to the group number by (3) so that consecutive cells within a block can be spread over multiple groups first. The upper bits are assigned to the bank and the bank address by (4) and (5). Our scheme can take advantage of a large packet that goes to one output queue. In this case, the cells belonging to the packet are spread over multiple groups first by (3). Since group accesses can be done in parallel without bank conflicts, this can minimize overall bank conflicts.

In this paper, we define an *output queue burst size* as the number of consecutive cells going to one output queue and use the term to characterize the input traffic behavior. Consecutive cells can go to one output queue for several reasons. First, a large packet that spans N cells in size enters the router. Then, we see N consecutive cells go to one output queue and the output queue burst size is N. Second, N small packets whose size are one cell long enter the router consecutively and they all go to the same output queue. In this case, the output queue burst size is also N. In both cases, our hashing scheme takes advantage of the burst and hashes consecutive cells to different groups first, which reduces bank conflicts. In this sense, the worst case bank conflicts can happen when the output queue burst size is 1. In this case, consecutive cells go to different output queues and then they all can be ended up in the same group and bank. Our simulations in later section assume the output queue burst size is 1 to simulate the worst case.

The cost of the hash function is very low because the operations in (1)–(5) are just bit-shifting, addition, and concatenation.

2) Bank Arbiter: We implemented two arbitration schemes for the bank aribiter. The first is longest queue first (LQF) and the second is longest latency first (LLF). Each bank FIFO is associated with a write and a read pointer. The difference between these two pointers gives the occupancy of the FIFO in terms of cells. LQF compares the occupancy of bank FIFOs in the same group and chooses one with the smallest occupancy. With a 32-entry bank FIFO, the occupancy register requires only 6 bits. The 6-bit comparators are not costly. For LLF implementation, we need a little more hardware support. Each bank FIFO entry is associated with a 16-bit enqueue time register. When a cell read/write request is enqueued into a bank FIFO, we save the global cycle counter value into the enqueue time register. When scheduling bank FIFOs, the LLF scheduler compares the enqueue time of cells at the head of the each bank FIFOs and chooses the bank with the largest cell latency. This is a little more costly compared to LQF.

3) QoS-Aware Scheduler (or Class Scheduler): The LQF optimizes the FIFO sizes. However, it does not optimize the latency. For instance, if a cell is enqueued into an almost empty FIFO while there are FIFOs whose occupancy is larger than the almost empty FIFO, the dequeue of the cell in the almost empty queue is delayed until all other FIFOs become almost empty. The LLF, on the other hand, tries to reduce the latency of cells too much, which results in an increasing chance of FIFO overflow. Thus these two are not a good candidate for the QoS-aware scheduler. From this point on, we will use QoS scheduler instead of QoS-aware scheduler for short.

The major challenge in QoS scheduler is to meet both latency and packet loss requirement assigned to each class. While the latency requirement is a latency bound that the memory controller should guarantee with a very high probability, the packet loss requirement is translated into a probability of packet loss. In packet memory controller, packet loss can happen for two reasons. First, it happens when the FIFO overflow happens. Second, when the memory controller does not meet the latency requirement for a latency sensitive packet, e.g., VoIP or video packet, the packet is dropped at the destination because it is no longer useful. For this reason, the QoS scheduler should be capable of meeting the latency requirement not to introduce additional packet loss other than FIFO overflows. Another challenge in the QoS scheduler design is to guarantee the maximum latency through the memory controller under dynamic traffic loads for different classes.

The proposed QoS scheduler is capable of two things. First, it guarantees the maximum cell latency through FIFO with a specified probability. Second, it is capable of adjusting scheduling weights for different classes dynamically as a response to the input load changes. Fig. 5 nicely depicts these two concepts. It shows a typical probability density function (PDF) of the cell latency through FIFO. This latency is called *FIFO latency* from this point on. The latency requirement is specified by two parameters M and N. M is a target latency threshold and N is the fraction of cells that have the cell FIFO latency larger than the target latency threshold. Thus, the line itself in PDF specifies the latency requirement. As the input load changes for a

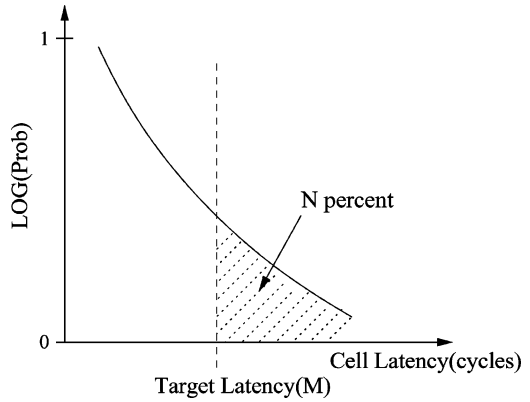


Fig. 5. PDF of FIFO latency distribution. Y-axis is logarithmic. The line specifies the latency requirement.

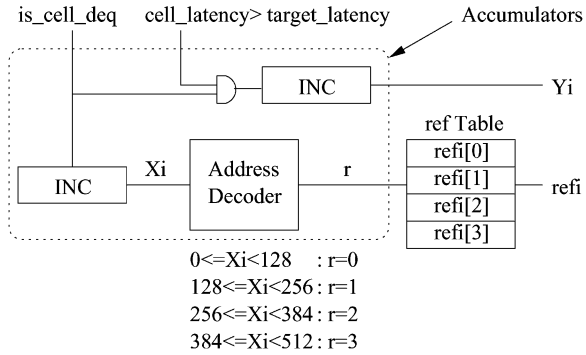


Fig. 6. Detailed diagram for accumulators and reference table.

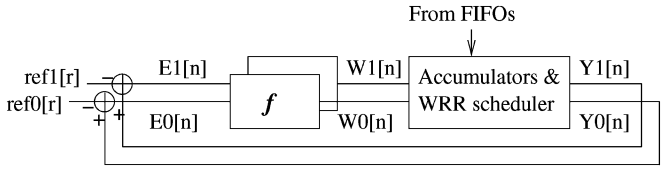


Fig. 7. QoS scheduler (or class scheduler).

certain class, the line shifts to the right or left as more or less latency violations happen. The QoS scheduler adjusts weights for classes according to the load changes and allocates more or less bandwidth to different classes so that the PDF lines for different classes stay at the same position.

To achieve this, the QoS scheduler collects two pieces of data every predetermined U cycles. This is called *weight update interval*. The first piece of data collected is the total number of cells dequeued from bank FIFOs for each class during the weight update interval. The second piece is the number of cells whose FIFO latency is larger than the target latency M. Let us use X and Y to represent these data, respectively. Then, using an equation given in (6), we detect the load change for class i

$$\mathrm{Error}_i = \frac{Y_i}{X_i} - N_i \tag{6}$$

$$X_i \times \mathrm{Error}_i = Y_i - X_i \times N_i \tag{7}$$

$$E_i = Y_i - X_i \times N_i. \tag{8}$$

If Y_i/X_i is larger than N_I, Error_i becomes positive and the scheduler detects increasing latency violations and increases the weight for class i. Otherwise, i.e., Error_i becomes negative, the scheduler detects less violations, and decreases the weight. One problem with this method is that the detection requires a division operation which is costly. To avoid that, we multiply both sides by X_i as in (7). We use E_i to simplify $X_i \times \mathrm{Error}_i$ as in (8). N_i is a fixed number and X_i is a function of the input load. While $X_i \times N_i$ requires a multiplication, it can be done by a table lookup. For this, we divide the maximum range for the X_i into multiple sub-ranges. This is shown in Fig. 6. In the example shown in Fig. 6, the X_i range is broken into four sub-ranges. Then, we multiply N_i by median values that represent each sub-ranges. These predetermined $X_i \times N_i$ are stored in the *ref* table. Given X_i, the address decoder in Fig. 6 produces the sub-range number r, which is used as an index for the table lookup. The address decoder in Fig. 6 can be implemented using several most significant bits (MSBs).

Fig. 7 shows the basic components of the QoS scheduler (or class scheduler), which is based on a feedback control loop. Each group requires two QoS schedulers: one for the write FIFOs and the other for the read FIFOs.

The first sub-block called *accumulators and weighted round robin (WRR) scheduler* performs two operations. The inputs to this sub-block are the FIFO latency of the dequeued cell that was at the head of each bank FIFO and the FIFO occupancies of all bank FIFOs. The detailed diagram for accumulators is shown in Fig. 6. The accumulators are counters, shown as INC in Fig. 6, which are incremented when the FIFO dequeue happens and store X_i and Y_i for every weight update interval, where i refers to the class. In Fig. 7, n refers to the current weight update interval. At the end of each weight update interval, we determine the sub-range, r, based on X_i as explained in the earlier paragraph. Y_i along with r is fed back for the comparison with ref_i, where ref_i is from the table lookup. r is omitted in Fig. 7 to avoid confusion. r is used to find a reference value, $\mathrm{ref}_i[r]$ for given X_i.

The QoS scheduler performs a weighted round robin algorithm among different classes. The WRR assigns different weights to different classes as the weights represent the bandwidth allocation for different classes. Once the weights are loaded into the weight counters for each class initially, the counters are decremented as cells are dequeued from the corresponding classes. Different classes are serviced initially in a round robin fashion. When the weight counter for a certain class becomes zero, the class is no longer serviced until either of two cases happens: all other weight counters become exhausted or no more cells are in the FIFOs of other classes, i.e., no traffic for other classes. In both cases, new weights are loaded into the weight counters and a new round starts. The WRR is a work-conserving algorithm.

Second, another major component of the QoS scheduler is a weight generator, shown as f in Fig. 7. This weight generator computes new weights for all classes every weight update interval. The basic operation of computing new weights is described next. The $\mathrm{ref}_i[r]$, which is quantized $X_i \times N_i$ in (8), is compared against $Y_i[n]$. The difference becomes a quantized

error $E_i[n]$. Depending on $E_i[n]$, $W_i[n-1]$, and $\mathrm{ref}_i[r]$, we update the weight $W_i[n]$ accordingly. A general update equation is shown

$$W_i[n] = W_i[n-1] + f\left(E_i[n], W_i[n-1], \mathrm{ref}_i[r]\right). \quad (9)$$

We choose the equation in (10) for the function f since it is simple to design when compared with other alternatives and it gives a good performance in simulations

$$f\left(E_i[n], W_i[n-1], \mathrm{ref}_i[r]\right) = \begin{cases} \alpha, & \text{if } E_i[n] > 0, \\ 0, & \text{if } E_i[n] = 0 \text{ or} \\ & \quad E_i[n] < 0 \text{ and } W_i[n-1] = 1 \\ -1, & \text{if } E_i[n] < 0 \text{ and } W_i[n-1] > 1 \end{cases} \quad (10)$$

where

$$\alpha = \left\lceil \frac{E_i[n]}{\mathrm{ref}_i[r]} \right\rceil \quad (11)$$

$$\simeq E_i[n] \gg m. \quad (12)$$

When the $E_i[n]$ is positive, it means the latency of too many dequeued cells are larger than the target latency. Thus, we increase the scheduling weight by α so that the class i can be scheduled more often. α is ideally defined as the error, $E_i[n]$, divided by the target value, $\mathrm{ref}_i[r]$, as in (11). This makes the weight increase proportional to the magnitude of the normalized error. To avoid a division operation, the division is approximated and implemented as a shift operation as in (12). m is the bit position of the most significant 1 in $\mathrm{ref}_i[r]$. For instance, if $\mathrm{ref}_i[r]$ is 00010100_2, m is 4 because bit 4 is the most significant 1. When $E_i[n]$ is 00110111_2 and $\mathrm{ref}_i[r]$ is 00010100_2, α becomes 00000011_2. Finding m and shifting operation can be implemented with simple AND gates and multiplexers. When the $E_i[n]$ is negative, it means that the latency of too few cells are larger than the target latency threshold. Thus, we decrease the scheduling weight by 1 so that the weight does not decrease suddenly to a small number after a large error. The minimum number for the weight is 1. Based on new weights, the QoS scheduler performs weighted round robin until the next weight update.

III. Tools for Evaluating Packet Memory Controller

A. Simulator Architecture

An event driven simulator, shown in Fig. 8, is developed to evaluate our proposed memory controller. The simulator consists of five major components: input traffic generator, hash function logic, schedulers, statistics collection, and event handler. Depending on the input load, core clock, memory clock frequency, and DRAM parameters, the event handler for the input traffic generator generates cells and they are hashed by the hash function logic and enqueued into FIFOs. The event handler for the schedulers schedules cells based on the scheduler type. The event handler for statistics collection collects various statistics for logging as well as computing average, variance, peak values at the specified intervals.

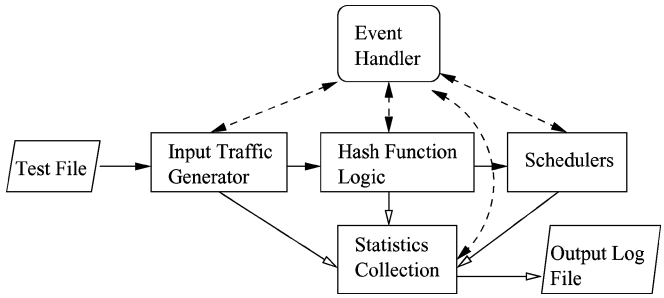


Fig. 8. Components of simulator.

Our simulator measures various characteristics such as FIFO occupancy, read or write FIFO latency, hashing performance, and FIFO occupancy and latency distribution under different configurations. Parameterized inputs to the simulator include the number of groups, number of banks per group, burst length of DRAM, DRAM clock frequency, core logic clock frequency, row cycle time (tRC) of DRAM, bank arbitration scheme, class scheduler type, input traffic burstness, input traffic load. For the QoS scheduler evaluation, additional inputs used are target latency threshold, latency violation target, weight update interval, input rate change intervals, and class ratios.

The outputs include average, variance, peak values for the FIFO occupancy, and FIFO latency. In the QoS scheduler evaluation, additional outputs produced are the sampled input rate variation, scheduling weight variation, FIFO occupancy variation, and FIFO latency variation for different classes over time.

B. Energy Model

We used the CACTI tool [18] to estimate the energy consumption for the SRAM FIFOs of the proposed memory controller. We evaluated the design using two different technologies: 90 and 65 nm. The results are shown in Section V-C.

IV. Experimental Results

A. Experimental Setting

Simulations are broken into three main categories. In the first category, we evaluate the performance of the hash function. In the second, we evaluate the performance of the single class version of our memory controller. In this configuration, the reorder buffer does not have a class scheduler and supports only single class traffic. In the third, we evaluate the performance of the multi-class version of our memory controller which includes the class scheduler (or QoS scheduler).

All test cases are created so that they can simulate the worst case behavior as in [4] and [5] for a fair comparison. For instance, maximum input load values are used in most of tests other than tests where we measure the effects of the different input load factors. Also, input patterns are designed so that they can introduce worst case bank conflicts.

The simulator generates cells based on the given load factor. When the cell is created, a random output queue number is assigned to the cell. The output queue burst size is 1 for all tests to simulate the worst case except for the hash performance test

(see Section IV-B) and the output queue burst size test (see Section IV-C-4). Using 1 for the output queue burst size means that a different output queue number is assigned for every cell created. It increases the probability of consecutive cells going to only one group and one bank, which creates the worst case FIFO occupancy and latency.

In the test cases for the QoS scheduler, control parameters are varied significantly so that the performance can be measured under the worst case. For instance, the ratio of input loads for two classes are varied from 1:9 to 9:1 to test the performance under a large input swing. Also, the input rate change interval is as small as 100 ns to simulate the rapidly changing inputs.

Assuming DDR DRAM parts, the DRAM burst size is 4 and the tRC for DRAM is eight memory clock cycles throughout the simulations. Using DDR DRAMs with the burst size of 4, a cell read or write can be issued to DRAM memories every two memory clock cycles.

All simulations were run to measure statistics for the 1 s period. In all simulations, average, variance, maximum FIFO occupancy, and FIFO latency are measured. The delay is measured in terms of memory clock cycles. Additional information such as scheduling weight variation, FIFO occupancy distribution, and FIFO latency distribution are measured in the QoS scheduler tests.

In the following sections, the simulation results for three categories are presented. First, Section IV-B discusses the performance evaluation for the hashing scheme. Second, Section IV-C presents various results from the simulations for the single class version of our memory controller. The results include comparison of LQF and LLF for the bank arbiter, optimization of bank FIFO structure for LQF, the effect of the output queue burst size for LQF, the FIFO occupancy, and FIFO latency distribution for LQF. Finally, Section IV-D shows the simulation results for the two-class version of our memory controller. The memory controller uses LQF as a bank arbiter because the latency requirement is guaranteed by the QoS scheduler (or class scheduler) and the LQF arbiter complements the class scheduler by optimizing the FIFO size. The results show how well the latency requirements are met by the proposed scheme, how well the scheme reacts to the dynamic input rate change for the different classes, and the effect of the input change rate.

B. Evaluating Hash Function Performance

The performance of the hashing scheme is evaluated from measuring the cell request distribution over multiple banks and groups. When a new cell request is generated in the simulation, an output queue number is randomly assigned to the cell. Each output queue has a running counter that keeps track of a cell sequence number which starts at 0 in the beginning. Depending on the output queue burst size, the same output queue number is assigned to the consecutive cell requests. This output queue number and cell sequence number for that output queue are mapped on a block address and a block offset, and they are mapped on the group, bank, and bank address by the hashing function. The average and variance of the cell requests seen by the reorder buffer for two output queue burst size (burst size $= 1$ and burst size $= 8$) are shown in Fig. 9. In this experiment, we varied the burst size from 1 to 16 but we show only two data point since two data points are sufficient to show the trends. The reorder buffer consists of eight groups and four banks per group and thus the total number of banks is 32. Good distribution among 32 banks is observed for the hashing function. The measured average requests for two burst sizes are almost same because roughly the same number of requests are served by each bank. The variance of requests indicates the burstness of the input traffic to each bank. A larger variance means a larger burstness. The variance for the burst size 8 is smaller than the burst size 1 because eight consecutive requests in the burst size of 8 are distributed over eight different banks while eight consecutive requests in the burst size of 1 can go to one bank in the worst case.

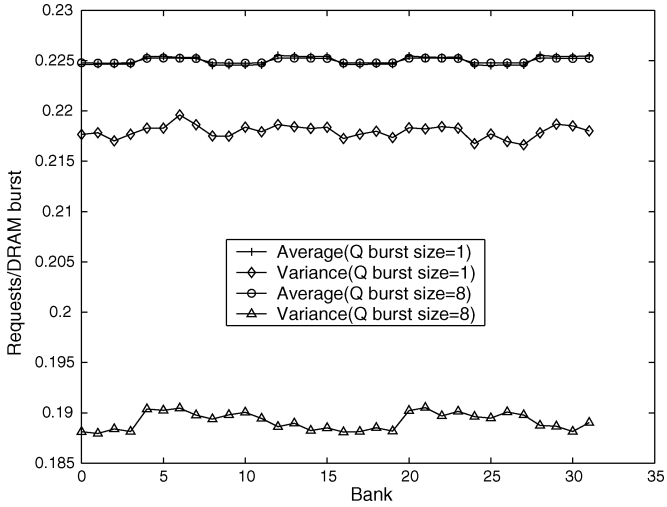


Fig. 9. Average and variance of cell read and write requests for 32 banks across 8 groups. Simulations are done for two output queue burst sizes, 1 and 8.

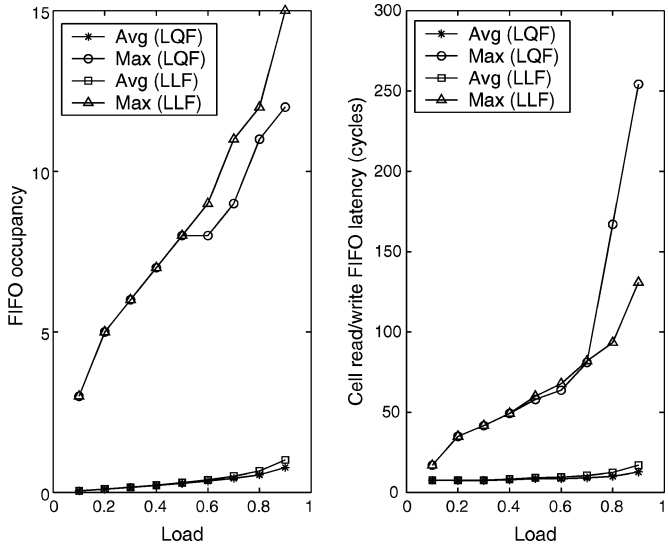


Fig. 10. Comparing average and maximum FIFO occupancies and cell read/write FIFO latencies through bank FIFOs for LQF and LLF arbiters (Groups $= 4$, Banks per group $= 8$).

C. Evaluating the Single Class Version of SQMC

1) Comparing LQF and LLF Bank Arbiters: Fig. 10 shows the measured average and maximum FIFO occupancies and de-

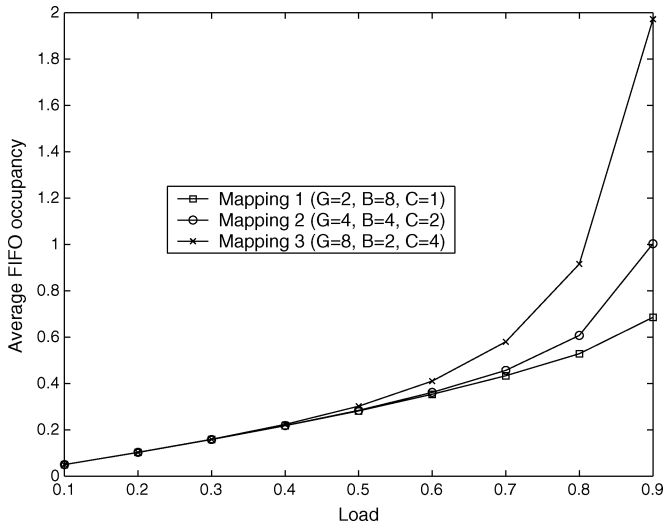


Fig. 11. Comparing the average FIFO occupancy for three different cell mappings.

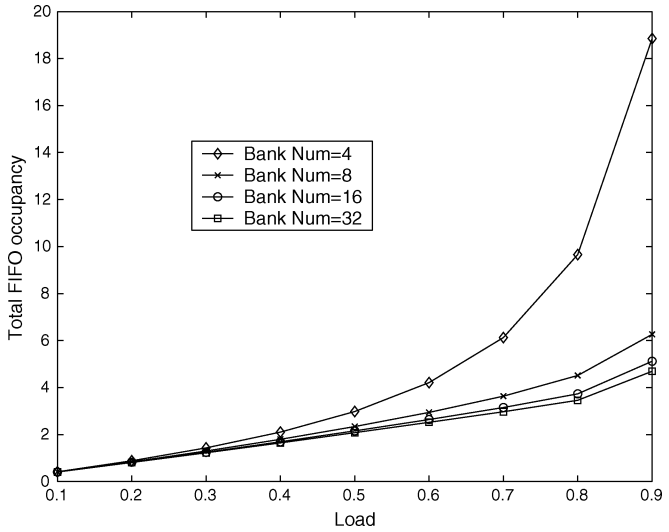


Fig. 12. Comparing total FIFO size requirements for different bank numbers (Groups = 4).

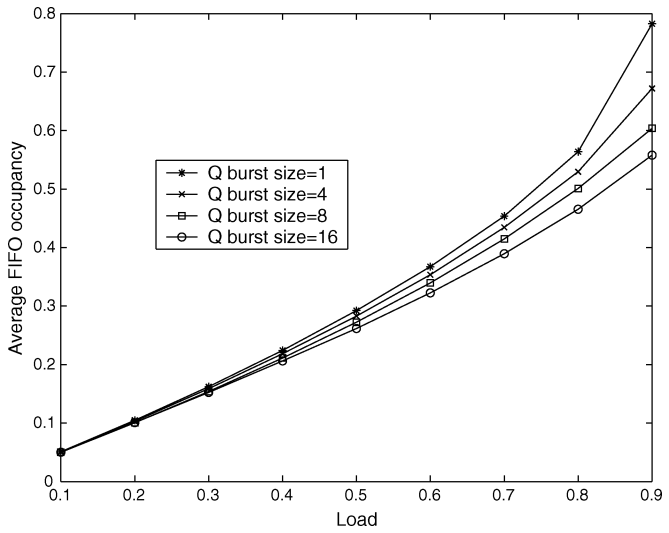


Fig. 13. Comparing the average FIFO occupancy for four output queue burst sizes, 1, 4, 8, and 16. (Groups = 8; Banks per group = 8).

lays of two arbitration schemes. The left figure compares the average and maximum FIFO occupancies for the LQF and LLF arbiter with respect to a load factor. The FIFO occupancy is measured per bank FIFO. The load factor varies from 0.1 to 0.9, where 0.9 means that the input traffic is equivalent to the 90% of the available memory bandwidth. In this simulation, the average FIFO occupancies are less than one entry for both arbiters whereas the maximum lengths reach 12 for LQF and 16 for LLF.

In the right side of Fig. 10, the average and maximum cell read/write FIFO latencies for two arbiters are compared. The LLF arbiter optimizes the maximum delay, not the average delay. Thus, the average delay for the LLF arbiter is slightly larger than that for the LQF arbiter whereas the maximum delay shows the opposite as the load gets close to 0.9.

In the rest of this paper, we show the results for the LQF arbiter as we use the LQF as the bank arbiter that is combined with the QoS scheduler in the multi-class version of our scheduler. This is because the QoS scheduler guarantees the latency requirements assigned to each class. As long as the latency is guaranteed by the QoS scheduler globally, the bank arbiter only needs to minimize the overall FIFO occupancy to avoid the FIFO overflow. LQF is best suitable for this.

2) Mapping Cells on DRAM Parts: For a given number of physical DRAM parts, the performance of hashing and reordering can vary depending on how to map a cell on physical DRAM parts. Fig. 11 shows the average FIFO occupancies for three different mappings. All three mappings use the same number of parts and have the same hardware complexity because the same number of bank FIFOs are required for all three mappings.

In mapping 1, two cells can be written or read simultaneously. The cell burst length of 1 takes one DRAM burst to read or write a cell. On the other hand, in mapping 3, it takes four DRAM bursts for one read or write transaction. In this case, eight cells can be simultaneously written into or read from the memory. From the results, we observe the following: it is better to spread a cell payload over many physical parts to reduce the access latency because it is less affected by bank conflicts.

3) Optimal Bank Numbers per Group: As the number of banks per group increases, its average FIFO occupancy becomes smaller. However, the total FIFO occupancy (average FIFO queue length $\times$ number of banks) does not decrease proportionally because of the increasing number of banks. As shown in Fig. 12, beyond the eight banks, the gain becomes marginal.

4) Effect of Different Output Queue Burst Sizes: Fig. 13 shows the effect of different output queue burst sizes. Since our hashing function well distributes cells with large output queue burst sizes as we have seen in Fig. 19, the average FIFO occupancy decreases as the cells become more bursty in terms of the output queue number.

5) FIFO Occupancy Distribution of LQF Arbiter: Fig. 14 shows the measured probability mass function of the FIFO occupancy distribution. The probability decreases exponentially

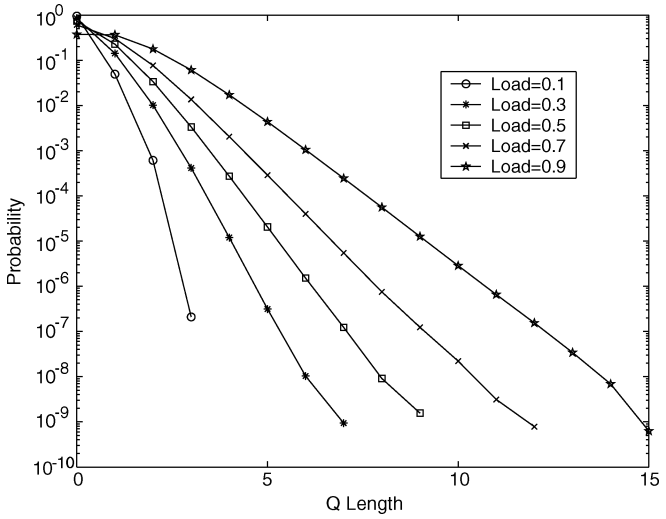


Fig. 14. Comparing FIFO occupancy distribution for different loads (Groups = 4; Banks per group = 8).

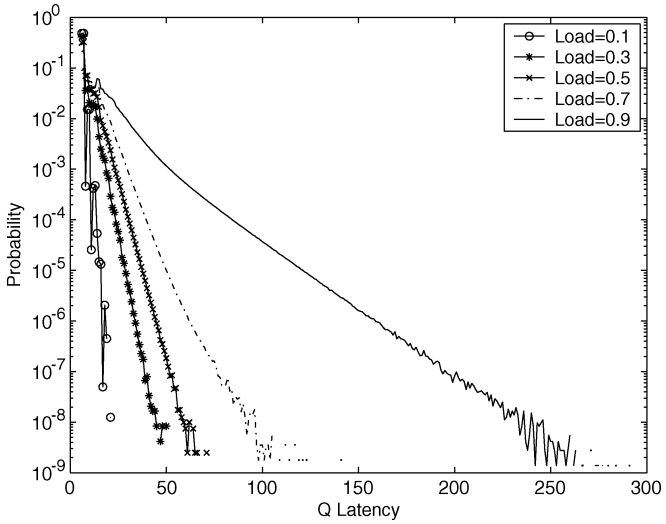


Fig. 15. Comparing FIFO latency distribution for different loads (Groups = 4; Banks per group = 8).

with the increasing FIFO occupancy. The slope gets steeper as the load gets smaller.

6) FIFO Latency Distribution of LQF Arbiter: Fig. 15 shows the probability density function of FIFO latency distribution. Again, the probability decreases exponentially with the increasing FIFO latency. As the load increases, more bank conflicts happen, which increases the FIFO occupancy and thus the latency of the cells in the FIFOs.

D. Evaluating a Multi-Class Version of SQMC

We implemented a multi-class version that supports two classes. As stated in the earlier section, we use LQF as the bank arbiter that is combined with the QoS scheduler. In these simulations, the main focus was to measure the performance of the QoS scheduler by showing how well the scheduler meets the latency requirements for two classes under varying traffic loads. The latency requirement is given in the following format: $N\%$ of cells have the FIFO latency larger than M cycles. From

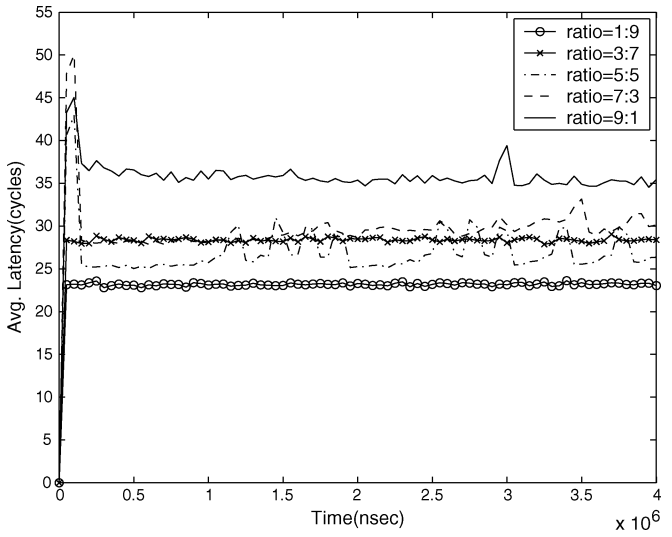


Fig. 16. Comparing average FIFO latency for five different class ratios (Groups = 4; Banks per group = 8).

Sections IV-D1–IV-D5, we assume that the class 0 has higher priority than the class 1 and a tight latency requirement is given to the class 0 whereas a loose one is given to the class 1. The latency requirement for class 0 in Sections IV-D1–IV-D5 is that 0.1% of cells have the FIFO latency larger than 60 cycles, whereas the requirement for class 1 is that 1% of cells have the FIFO latency larger than 400 cycles. The requirements are carefully chosen so that the total aggregate bandwidth required to meet the latency requirements is close to 0.9 of the total available memory bandwidth. This is done to introduce a severe congestion and see the benefit of the QoS scheduler.

In these simulations, the weight update interval for the dequeue weight change is set to 100 μs. That is, every 100 μs, we sample the latency distribution and adjust the scheduling weight for class 0 and 1 if necessary. 100 μs is chosen for the weight update interval because the interval shorter than that gives too small number of dequeued cells and the interval longer than that acts slowly to the input rate change. For the number of sub-ranges for X_i in Figs. 6 and 10 is used. As the number increases, the quantization errors become smaller. But, our experiments show that for larger than 10 sub-ranges, the gain was marginal.

In the first set of tests (constant input load case), we vary the ratio of input loads for class 0 and 1 from 1:9 to 9:1. Once the ratio is set initially, it remains the same throughout the simulation in these tests. In the second set of tests (dynamic input load case), the ratio of the input loads between class 0 and 1 changes dynamically in the middle of simulation to see how well the scheduler tracks the traffic variations.

1) Average FIFO Latency for the Constant Input Load Case: Fig. 16 shows the average FIFO latency of cells for five different class ratios (class0:class1), 1:9, 3:7, 5:5, 7:3, and 9:1. The total aggregate load for class 0 and 1 is set to 0.9 of the available memory bandwidth. Thus, 1:9 implies that the input load for class 0 is 0.09 and the one for the class 1 is 0.81. For the 1:9 ratio case, the average latency is also small as the FIFO occupancies are small. For medium input loads from 3:7 to 7:3, the

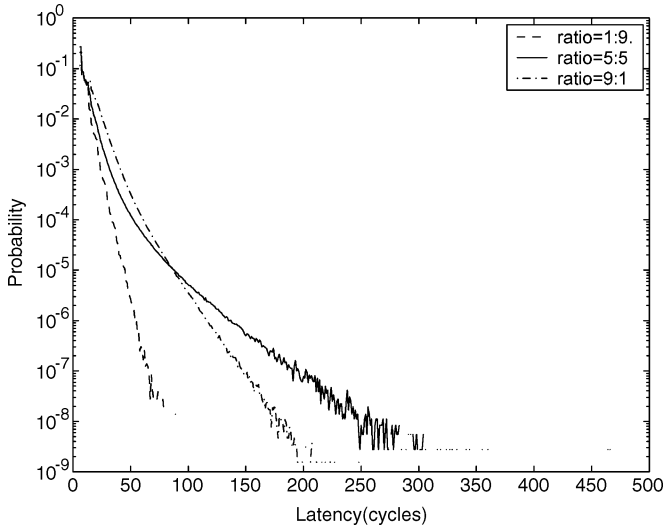


Fig. 17. Distribution of FIFO latency for class 0 with three different class ratios (Groups $= 4$; Banks per group $= 8$ constant load).

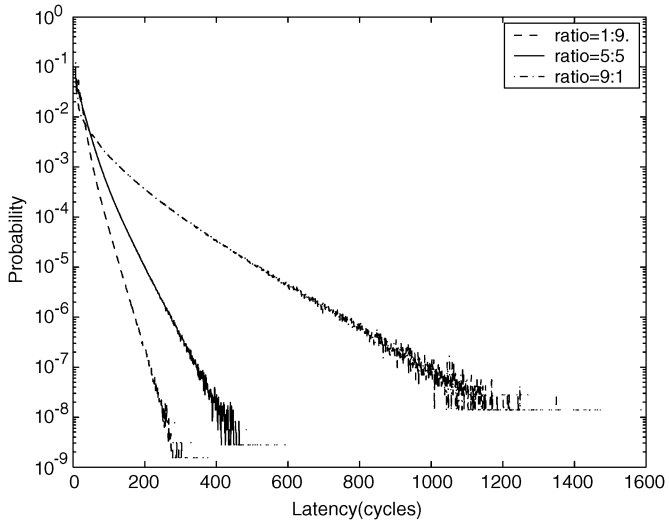


Fig. 18. Distribution of FIFO latency for class 1 with three different class ratios (Groups $= 4$, Banks per group $= 8$, constant load).

latencies are roughly 28 cycles. For the 9:1 ratio case, the latency increases to 35 cycles because more cells have larger FIFO latencies as shown in Fig. 17.

2) PDF of FIFO Latency Distribution for the Constant Input Load Case: Figs. 17 and 18 show the PDF of FIFO latency for three different class ratios 1:9, 5:5, and 9:1. The latency requirement for the class 0 is that 0.1% of the cells have the FIFO latency larger than 60 cycles. One interesting point in Fig. 17 is that there is a crossover point between 5:5 and 9:1 around 80 cycles. In the 9:1 case, the scheduling weight for the class 0 stays high almost always, which leads to less number of violations beyond the 80 cycles. Meanwhile, in the 5:5 case, the scheduling weight for the class 0 fluctuates due to relatively a small input load. This introduces more numbers of violations beyond the 80 cycles when the weight is small. This finding is consistent in Fig. 18. Although the input load for the class 1 decreases in the order of 1:9, 5:5, and 9:1, 9:1 has the worst performance. This is because the QoS scheduler aggressively allocates large

TABLE I
ACHIEVED N FOR CLASS 0 WITH VARYING INPUT LOAD RATIO

C0:C1 ratio	1:9	3:7	5:5	7:3	9:1
constant load	0.0002%	0.021%	0.094%	0.098%	0.13%
dynamic load	0.085%	0.071%	0.094%	0.095%	0.12%

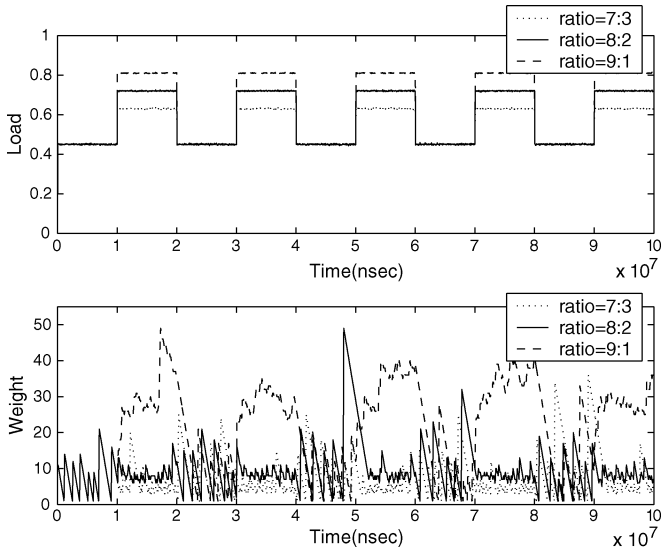


Fig. 19. Input load and scheduling weight for class 0 with three different class ratios (Groups $= 4$; Banks per group $= 8$, dynamic load).

bandwidth for class 0 to meet the tight latency requirement of class 0, which leaves small bandwidth for class 1 as expected. Table I numerically shows how well this latency requirement for the class 0 is met by the QoS scheduler for five different ratios. The second row of table shows the achieved value for N in Fig. 5 for the constant input load case. All ratios except for the 9:1 achieve less than 0.1%. Severe load for class 0 in the 9:1 case results in 0.13%, which is slightly larger than the target N.

3) Weight Change for the Dynamic Input Load Case: Fig. 19 shows the input load and scheduling weight changes for class 0 over 100 ms of the simulation time. Three lines represent three different input load ratios between class 0 and 1. The top plot in Fig. 19 shows the input loads of class 0 for three ratios. Initially, the ratios between class 0 and 1 are 5:5 for all three lines and after 10 ms they become 7:3, 8:2, and 9:1, respectively. During the periods where ratios are 5:5, the weights for three ratios fluctuate as the weight increase upon large errors suppresses the errors in the following cycles. This is shown in the bottom plot. As soon as the ratios change to 7:3, 8:2, and 9:1, the weights track the changes to meet the latency requirements. They do not vary much compared to the 5:5 periods because weights remain high to meet the latency requirements aggressively, which regulates the error rates. The weight for the class 1 almost does not change much since it meets its loose latency requirement and so it is not shown here.

4) PDF of FIFO Latency Distribution for the Dynamic Input Load Case: Fig. 20 shows the PDF of the latency distribution for class 0. Three lines represent again PDF for three different ratios, 7:3, 8:2, and 9:1. Again, initially the ratios for all three lines are 5:5. They become 7:3, 8:2, and 9:1 after 10 ms. Every 10 ms, the input load ratios alternate between 5:5 and 7:3, 8:2,

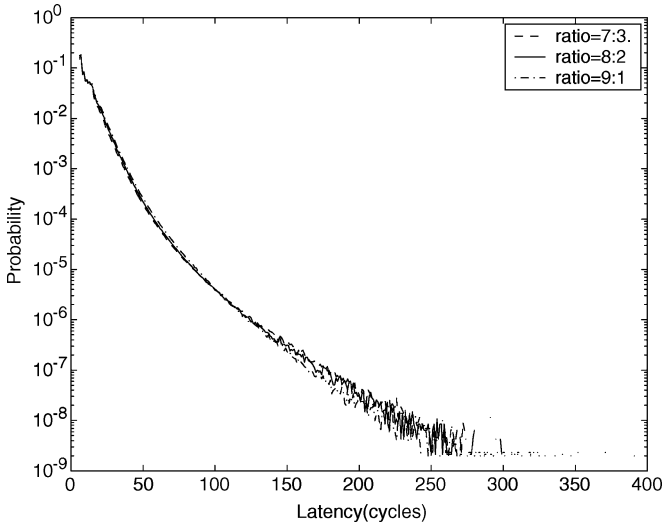


Fig. 20. Distribution of FIFO latency for class 0 with three different class ratios (Groups $= 4$, Banks per group $= 8$, dynamic load).

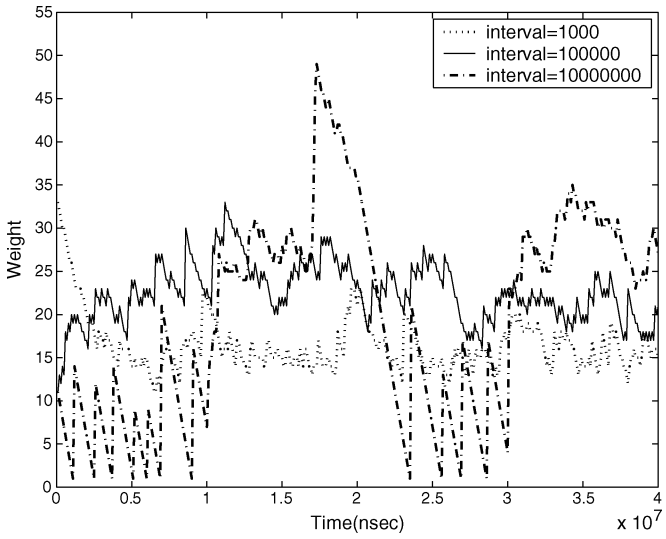


Fig. 21. Scheduling weight for class 0 with three different input load change intervals (Groups $= 4$, Banks per group $= 8$).

or 9:1. In Fig. 20, all three lines are almost overlapped with each other. They are pretty close to the latency distribution of 5:5 in Fig. 17. This is because during the half of simulation time, input load ratios are 5:5 and performance in these periods dominates the distribution. The third row of Table I shows the achieved value for N in Fig. 5 for the dynamic input load case. Compared with the static input load case in the second row, the values are closer to 0.1% since the periods with 5:5 ratios dominate the performance.

5) Response to Rapidly Changing Input Load: One of the key metrics when evaluating the QoS scheduler is how well it tracks the rapidly changing inputs. In this simulation, we vary the input load change interval from 1000 to 10 000 000 ns while we fix the weight update interval to 100 μs. Again the input load ratio for class 0 and 1 alternates between 5:5 and 9:1. Figs. 21 and Fig. 22 show the weight changes for class 0 and the PDF of the latency distribution for class 0, respectively. For the case

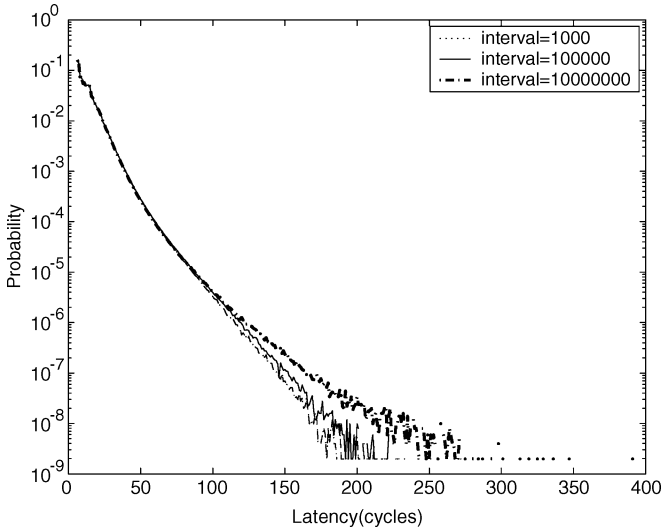


Fig. 22. Distribution of FIFO latency for class 0 with three different input load change intervals (Groups $= 4$, Banks per group $= 8$).

TABLE II
ACHIEVED N FOR CLASS 0 WITH VARYING INPUT LOAD CHANGE INTERVALS

interval	1 μs	10 μs	100 μs	1 ms	10 ms
	0.11%	0.11%	0.12%	0.12%	0.12%

where the input load change interval is 1000 ns, the weight ranges from 10 to 25 throughout the simulation as shown in Fig. 21. For 100 000 ns case, the weight ranges from 15 to 30. These indicate that the feedback scheduler acts as a low pass filter for the rapidly changing input load to meet the latency requirement. The scheduler performance is shown in Fig. 22 and Table II. Table II shows that the scheduler performs consistently well with varying input load change intervals.

V. ANALYSIS

A. Overflow Probability

The proposed scheme does not guarantee no overflow. However, its probability decreases significantly as the bank FIFO size increases as shown in Fig. 14. Overflow upon writing cells translates into packet loss. However, a small buffer with a little bit of speedup in the data path makes it possible to avoid packet loss upon overflow considering a very small overflow probability. For instance, if a bank FIFO is full when a cell needs to be written into the FIFO, it can be stored in a small buffer. Assuming reading from the bank FIFO has a speedup over writing into it and overflow is transient, the speedup will free the FIFO entries and the buffered cell will be processed by the FIFO without loss. Overflow upon reading cells translates into delaying the access by stalling the read scheduler. Again, a little speedup can make this delay transient.

To compute the probability of overflow, we may derive a queueing model and use the Chebychev bound to compute the overflow probability but the bound is not tight enough to give any meaningful information. Instead, we measure the distribution from simulation and estimate the probability by projecting the lines in Fig. 14. For load $= 0.9$, 1.5 additional FIFO entry

TABLE III
COMPARING SRAM AREAS FOR TWO SCHEMES

line-rate	RADS	SQMCv1	SQMCv2
OC-768	$128 \sim 600KB$	$24.9KB$	$33.2KB$
OC-3072	$4MB \sim 10.5MB$	99.6 KB	$132.8KB$

TABLE IV
SRAM AREAS FOR TWO-CLASS VERSION OF SQMC

line-rate	SQMCv3	SQMCv4
OC-768	$51.3KB$	$68.4KB$
OC-3072	205.1 KB	$273.5KB$

TABLE V
COMPARING THE POWER CONSUMPTION BY SRAM FIFOS FOR 90- AND 65-nm TECHNOLOGY

Technology	SQMCv1	SQMCv2
90 nm	71.6 mW	80.64 mW
65 nm	33.52 mW	40.16 mW

drops roughly the overflow probability by one decade. With a 24-entry deep bank FIFO (SQMCv1), the probability will be roughly 10^{-15}. With a 32-entry deep bank FIFO (SQMCv2), the probability will be roughly 10^{-20}. Core routers should be very robust in terms of packet loss. One year is translated into 1.26×10^{16} memory clock cycles assuming the memory clock runs at 400 MHz. Since the FIFO occupancy was logged every two memory clocks, 10^{-15} overflow probability means we lose a cell every 1.90 month per memory controller. Considering that, achieving 10^{-5} less probability in SQMCv2 by allocating 33% more area seems to be a reasonable tradeoff. Both SQMCv1 and SQMCv2 are a single class version of our memory controller. For the multi-class version, bank FIFOs need to be duplicated. Since the input load is shared by multiple classes, the input load seen by one class's bank FIFO becomes smaller. Thus, the overflow probability becomes even smaller for the multi-class version.

As stated earlier, the 64-byte cell size was used throughout the simulation for a fair comparison with previous works. However, our results are still valid for bigger cell sizes. For instance, for a twice bigger cell size, 128 bytes, the load seen by the bank FIFO becomes half because it takes twice longer time to transfer the payload. But, the FIFO entry size needs to be doubled. Thus, the roughly the same area is required for FIFOs to achieve the same overflow probability.

In our method, packet loss, i.e., overflow probability, is a function of the FIFO size and the input load. As the line-rate increases, as long as we manage the input load per bank FIFO same, the overflow probability remains the same for a given FIFO size. To keep the input load same, we need to increase the number of groups proportional to the line-rate increase.

B. Area

In Table III, RADS is the scheme reported in [4], SQMCv1 is our scheme with 24-entry deep bank FIFOs, and SQMCv2 is with 32-entry deep bank FIFOs. Both SQMCv1 and SQMCv2 are a single-class version of our memory controller. This was done for a fair comparison because RADS was supporting only a single class. The SRAM areas used to implement the FIFOs in our scheme and in [4] are compared in Table III. RBAU in [4] is reported to take roughly four times less area than RADS for OC-768 and OC-3072. Our SMCv1 is better than RBAU by an order in OC-3072. More importantly, our scheme works for hundreds of thousands of output queues without any additional area penalty whereas their schemes will be forced to increase the SRAM area significantly and may not be feasible.

For a multi-class version of our memory controller, we need to duplicate the bank FIFOs by the number of classes and add area for the enqueue time registers. The results are shown in Table IV. SQMCv3 is a two-class version with 24-entry deep bank FIFOs and SQMCv4 is a two-class version with 32-entry deep bank FIFOs. They take roughly twice the areas of SQMCv1 and SQMCv2, respectively.

A formula to compute the SRAM areas for our schemes is shown in (13). The structure of the SRAM area consists of three major components: write request FIFOs, read request FIFOs, and read data buffer. The formula for the area is broken into these three components. In (13), C, G, B, and E stand for the number of classes, groups, banks, and FIFO entries, respectively. $enq.reg$ represents the bit width of an enqueue time register. 512 (in bits) and 19 are used for data and addr, respectively. $enq.reg$ is 0 for a single-class version and 16 for a two-class version. C is 1 for a single-class version and 2 for a two-class version. G is 1 for OC768 and 4 for OC3072. B is 8. E is 24 for SQMCv1 and SQMCv3 and 32 for SQMCv2 and SQMCv4

$$\begin{aligned}\text{Area} = &\overbrace{(data + addr + enq.reg) \times C \times G \times B \times E}^{\text{write request FIFO}} \\ &+ \overbrace{(addr + enq.reg) \times C \times G \times B \times E}^{\text{read request FIFO}} \\ &+ \overbrace{data \times C \times G \times B \times E}^{\text{read data buffer}}. \end{aligned} \tag{13}$$

We compared only SRAM areas for different schemes because it is a dominant factor in terms of die area. Custom logic in our memory controller scheme contains small adders, shifters, and logic gates, which are relatively cheap. Our scheme and previous work use the same packet memory size.

C. Power Consumption

We estimated the power consumption for SQMCv1 and SQMCv2 supporting OC-3072 by using the CACTI tool. We assume the memory clock runs at 400 MHz. The results are shown in Table V. If the technology is scaled from 90 to 65 nm, the power consumptions are decreased roughly by a factor of two.

As the line-rate increases further beyond OC-3072, we need to increase the number of groups proportionally to meet the bandwidth requirement and expect the power will go up proportionally as the power consumption is proportional to the number of groups.

VI. Conclusion

In this paper, we have proposed a packet memory controller that is scalable beyond OC-3072 and provides QoS to packets with different latency (or QoS) requirements. This scheduler takes much less SRAM area compared to existing schemes whereas it virtually does not suffer the packet loss problem. Our scheme well supports a large number of output queues which is critical in the modern routers. In addition, the QoS support in our scheme becomes critical as more internet packets become latency sensitive.

References

[1] C. Minkenberg, R. Luijten, W. Denzel, and M. Gusat, "Current issues in packet switch design," in *Proc. HotNets-I*, 2002, pp. 119–124.

[2] R. Ramaswami and K. N. Sirvajan, *Optical Networks*. San Mateo, CA: Morgan Kaufman, 1990.

[3] S. Iyer, R. Kompella, and N. McKeown, "Designing buffers for router line cards," Stanford Univ., Stanford, CA, Tech. Rep. TR02-HPNG-031001, 2002 [Online]. Available: http://klamath.stanford.edu/ sundaes/publications.html

[4] J. García, M. March, L. Cerdá, J. Corbal, and M. Valero, "A DRAM/SRAM memory scheme for fast packet buffers," *IEEE Trans. Comput.*, vol. 55, no. 5, pp. 588–602, May 2006.

[5] J. García, J. Corbal, L. Cerdá, and M. Valero, "Design and implementation of high-performance memory systems for future packet buffers," in *Proc. MICRO*, Dec. 2003, pp. 372–384.

[6] A. Nikologiannis and M. Katevenis, "Efficient per-flow queueing in DRAM at OC-192 line rate using out of order execution techniques," in *Proc. IEEE Int. Conf. Commun.*, 2001, pp. 2048–2052.

[7] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, "Memory access scheduling," in *Proc. 27th Int. Symp. Comput. Arch.*, 2000, pp. 128–138.

[8] M. Valero, T. Lang, M. Peiron, and E. Ayguade, "Increasing the number of conflict-free vector access," *IEEE Trans. Comput.*, vol. 44, no. 5, pp. 634–646, May 1995.

[9] Fujitsu. Tokyo, Japan, "256 M bit double data rate FCRAM," MB81N26847B/261647B-50/-55/-60 data sheet [Online]. Available: http://www.fujitsu.com, 2004

[10] Infineon Technologies. Munich, Germany, "RLDRAM. High density, high-bandwidth memory for networking applications," [Online]. Available: http://www.infineon.com 2004

[11] W. Aly and H. Lutfiyya, "Using feedback control to manage QoS for clusters of servers providing service differentiation," in *Proc. IEEE GLOBECOM*, Nov. 2005, pp. 960–964.

[12] Y. Lu, T. Abdelzaher, and C. Lu, "Feedback control with queueing-theoretic predicition for relative delay guarantees in web servers," in *Proc. 9th IEEE Real-Time Embedded Technol. Appl. Symp.*, May 2003, pp. 208–217.

[13] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic, "ControlWare: A middleware architecture for feedback control of software performance," in *Proc. 22nd Int. Conf. Distrib. Comput. Syst.*, Jul. 2002, pp. 301–310.

[14] T. Abdelzaher, K. Shin, and N. Bhatti, "Performance guarantees for web server end-systems: A control-theoretical approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 1, pp. 80–96, Jan. 2002.

[15] C. Lu, T. Abdelzaher, and J. Stankovic, "A feedback control approach for guaranteeing relative delays in web servers," in *Proc. IEEE Real-Time Technol. Appl. Symp.*, Jun. 2001, pp. 51–62.

[16] D. Hoang, "Application of control theory in QoS control," in *Proc. ICACT*, Feb. 2006, pp. 896–901.

[17] The Internet Engineering Task Force, "An architecture for differentiated services," RFC2475, 1998 [Online]. Available: http://tools.ietf.org/html/rfc2475

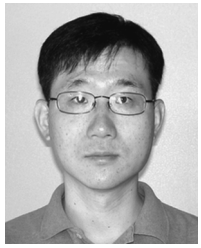
[18] D. Tarjan, S. Thoziyoor, and N. Jouppi, "CACTI 4.0," HP Laboratories, Palo Alto, CA, Jun. 2006 [Online]. Available: http://www.hpl.hp.com/techreports/2006/HPL-2006-86.html

[19] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing router buffers," in *Proc. ACM SIGCOMM*, Aug. 2004, pp. 281–292.

[20] D. Wischik and N. McKeown, "Part I: Buffer sizes for core routers," *Comput. Commun. Rev.*, vol. 35, no. 3, pp. 75–78, 2005.

[21] A. Dhamdhere, H. Jiang, and C. Dovrolis, "Buffer sizing for congested internet links," in *Proc. IEEE INFOCOMM*, Mar. 2005, pp. 1072–1083.

[22] A. Kloth, *Advanced Router Architecture*. Boca Raton, FL: CRC, 2005.



Hyuk-Jun Lee (S'94–M'04) received the B.S. degree in computer science and engineering from the University of Southern California, Los Angeles, in 1993 and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1995 and 2001, respectively.

Currently, he works with Cisco Systems, San Jose, CA. During his leave from 1995 to 1996, he worked at Swan Instruments, Santa Clara, CA. His research interests include computer architecture and arithmetic, VLSI design, network and communication algorithms.

Eui-Young Chung (M'06) received the B.S. and M.S. degrees in electronics and computer engineering from Korea University, Seoul, Korea, in 1988 and 1990, respectively, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 2002.

Currently, he is an Associate Professor with the School of Electrical and Electronic Engineering, Yonsei University, Seoul, Korea. From 1990 to 2005, he was a Principal Engineer with SoC R&D Center, Samsung Electronics, Kiheung, Korea. His research interests include system architecture and VLSI design including all aspects of computer aided design with the special emphasis on low power applications and in the design of mobile systems.