Design of a bitmap-based QoS-aware memory controller for a packet memory

Seunghak Yu[1,2], Sungroh Yoon[2], Eui-Young Chung[3], and Hyuk-Jun Lee[4a)]

[1] *Department of IT Convergence, Korea University, 145 Anam-ro, Seongbuk-gu, Seoul 136–701, Korea*

[2] *Department of Electrical and Computer Engineering, Seoul National University, 1 Gwanak-ro, Gwanak-gu, Seoul 151–744, Korea*

[3] *Department of Electrical and Electronic Engineering, Yonsei University, 134 SinChon-dong, Seodaemun-gu, Seoul 120–749, Korea*

[4] *Department of Computer Engineering, Sogang University, 1 Sinsu-dong, Mapo-gu, Seoul 121–742, Korea*

a) *hyukjunl@sogang.ac.kr*

Abstract: A packet memory controller in routers accesses the packet memory according to the QoS requirements of packets. The previous QoS-aware controller using a feedback control loop degenerates into round robin scheduling under temporary overload and suffers from slow response. We propose a new packet memory controller that estimates input load accurately and rapidly and schedules different classes using a flexible bitmap scheduler. The results show that under temporary overload or rapidly changing input loads, it can successfully meet the latency requirements by showing only less than 2% difference from the requirement of the high priority class.

Keywords: high-performance memory system, memory controller, packet memory

Classification: Electron devices, circuits, and systems

References

[1] H. Lee and E. Chung: IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **16** [3] (2008) 289.
[2] K. Nesbit, N. Aggarwal, J. Laudon and J. Smith: MICRO (2006) 208.
[3] N. Rafique, W. Lim and M. Thottethodi: PACT (2007) 245.
[4] O. Mutlu and T. Moscibroda: MICRO (2007) 146.
[5] Y. Kim, M. Papamichael, O. Mutlu and M. Harchol-Balter: MICRO (2010) 65.
[6] M. J. Flynn: *Computer Architecture: Pipelined and Parallel Processor Design* (Jones and Bartlett, Boston, 1995) 365.
[7] CISCO: The Cisco QuantumFlow Processor: Cisco's Next Generation Network Processor (2008) http://www.cisco.com.

© IEICE 2014
DOI: 10.1587/elex.11.20130983
Received December 13, 2013
Accepted February 04, 2014
Publicized February 17, 2014
Copyedited March 10, 2014

1 Introduction

Rapid deployment of high-performance core routers makes real-time internet applications such as streaming video service, video calls, Voice over Internet Protocol (VoIP), and online gaming more accessible. Explosive use of smartphones further increases these traffics. These applications typically have various QoS requirements, e.g. latency requirement, and packets should be classified and processed accordingly based on their requirements. A packet memory controller in routers is responsible for writing and reading packets, which are broken into fixed sized cells (e.g. 64 to 256 bytes), and should meet the latency requirements of different classes during writing and reading. The requirement is expressed as a latency distribution of cells in which the number of cells larger than the specified latency target is constrained [1], e.g. thes probability of cells whose memory access latency is larger than 200 cycles is 10^{-5}.

The existing QoS-aware controller using a feedback control loop degenerates into round robin scheduling under temporary overload and suffers from very slow response [1]. To resolve these issues, we propose a new method based on input load estimation and bitmap scheduling. The main contributions of this paper are as follows. First, the proposed method models the scheduler as a M/D/1 queue [6], which makes it possible to estimate the current input load for a given class accurately and determine stability based on the allocated memory bandwidth for that class. Thus, it can optimize scheduling weights for different classes to meet latency requirements even in temporary overload by giving proper scheduling weights to higher priority classes. Second, even with a small sampling window size, it can estimate the input loads accurately and change its scheduling weights rapidly because it does not depend on a feedback mechanism. Third, the bitmap scheduler provides very flexible scheduling patterns including M/D/1 with only small area penalty.

2 Related works

QoS-aware memory controllers were proposed in various contexts including a packet memory environment [1] and multi-processor environments [2, 3, 4, 5]. In [1], the proposed adaptive feedback mechanism dynamically adjusts allocated bandwidths to different classes based on latency violations. In [2, 3], a fair queueing method is employed to allocate bandwidth for different processor threads whereas in [4, 5], priority scheduling is used to schedule threads based on their sensitivity to inter-thread interference, latency, or bandwidth.

However, the methods proposed for multi-processor environment in [2, 3, 4, 5] are not adequate for the packet memory because data mapping on DRAM and data access patterns in the packet memory are different than those in the multi-processor environment. In [4, 5], they provide high priority to the threads with latency sensitivity. However, priority scheduling cannot simultaneously meet latency requirements of multiple classes even under non-

© IEICE 2014
DOI: 10.1587/elex.11.20130983
Received December 13, 2013
Accepted February 04, 2014
Publicized February 17, 2014
Copyedited March 10, 2014

overload cases.

The adaptive feedback mechanism in [1] uses a feedback control loop that adaptively changes scheduling weights for different classes based on latency violations. That is, the number of cells that violate the latency requirement is counted per class and the scheduling weight for the associated class gets incremented if the number of violated cells for the class is larger than the threshold value. If multiple classes have violations simultaneously, weights are continuously incremented and scheduling degenerates into round robin scheduling, which happens whenever inputs are overloaded temporarily. In addition, the method relies on a feedback mechanism which counts the number of cells violating the latency requirement per sampling window whiling not paying attention to the input load. This makes the sampling window relatively big to avoid sampling errors and slows down the response of the feedback mechanism, which leads to huge violations under rapidly changing input loads.

3 Background

3.1 Packet memory controller

Fig. 1 shows the packet memory controller in [1]. We refer it as a scalable QoS memory controller (SQMC). A typical packet memory controller consists of hash logic and reorder buffers. The hash logic takes the read or write addresses of continuous cells as input and assigns each a new address, thus distributing them to multiple banks and parts. The reorder buffer architecture consists of bank FIFOs, bank arbiters, class schedulers, and a read/write arbiter.

In this work, the QoS scheduler is replaced by the proposed bitmap scheduler (BMS). A bitmap scheduler refers to a type of scheduler that creates a bitmap from the scheduling pattern based on the individual class weights.

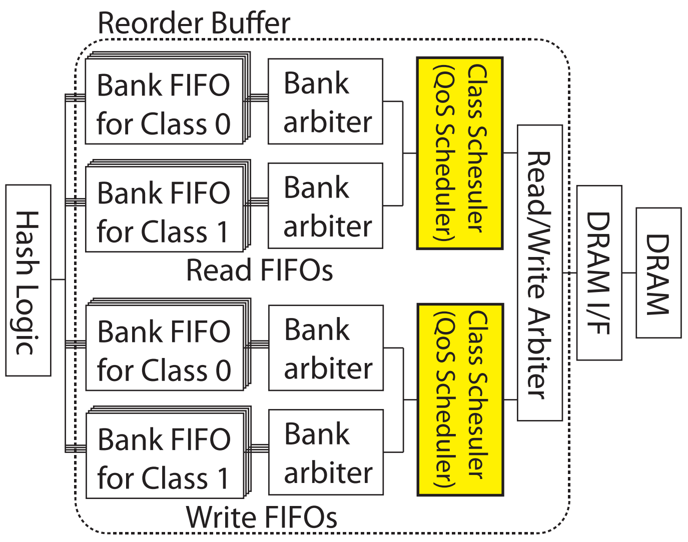

Fig. 1. Architecture of packet memory controller [1]. This paper proposes a new bitmap-based QoS scheduler.

© IEICE 2014
DOI: 10.1587/elex.11.20130983
Received December 13, 2013
Accepted February 04, 2014
Publicized February 17, 2014
Copyedited March 10, 2014

One advantage of using a bitmap scheduler is that, for the same weight configuration, a bitmap scheduler can generate different bitmap patterns to accommodate specific user needs.

4 Proposed method

4.1 Overview

Fig. 2 shows the overall flow of the proposed bitmap scheduler. As a system requirement, a latency requirement is given for each class in the beginning. In off-line design time, we first model the QoS scheduler for a single logical bank[1] as an M/D/1 queue and then pre-calculate the memory bandwidth allocation needed for each class with respect to different input load values. For instance, to meet a given requirement of high priority (class 0), 10% of total available memory bandwidth should be allocated for the input load of 0.1 and 20% for the input load of 0.2, and so forth. Next, we determine the scheduling weights for different classes to achieve the pre-calculated class bandwidths. For instance, to allocate 20% and 40% of total available memory bandwidth for high and low priority class respectively, we use a weight ratio of 1:2 for two classes. Then, we create a bitmap using the class weights and then optimize them to generate a two-level lookup table. This off-line procedure and hardware implementation are explained from section 4.2.1 through 4.3. Later in runtime, the scheduler measures the input loads and schedules different classes by indexing the bitmap table using the measured input load values. The rest of this section explains the further details of each step.

4.2 Offline scheduler

4.2.1 M/D/1 modeling of QoS scheduler

We can model the QoS scheduler as an M/D/1 queue due to the following properties. First, the hash function randomizes mapping a cell to logical banks and parts. This randomization makes incoming cells seen by a logical

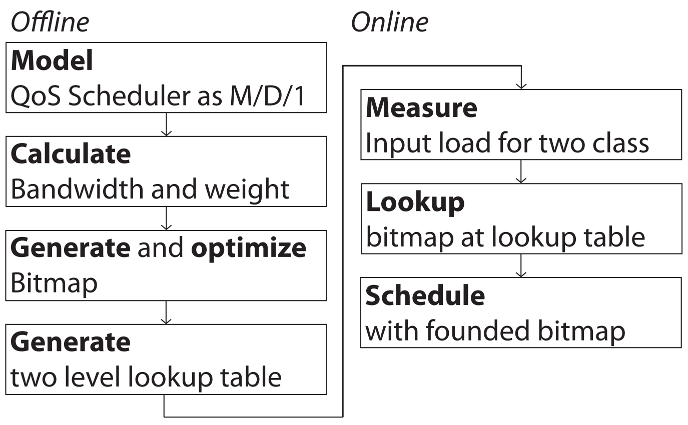

Fig. 2. Overall flow of proposed bitmap scheduler.

[1]A logical bank is a group of banks where a cell is mapped to.

© IEICE 2014
DOI: 10.1587/elex.11.20130983
Received December 13, 2013
Accepted February 04, 2014
Publicized February 17, 2014
Copyedited March 10, 2014

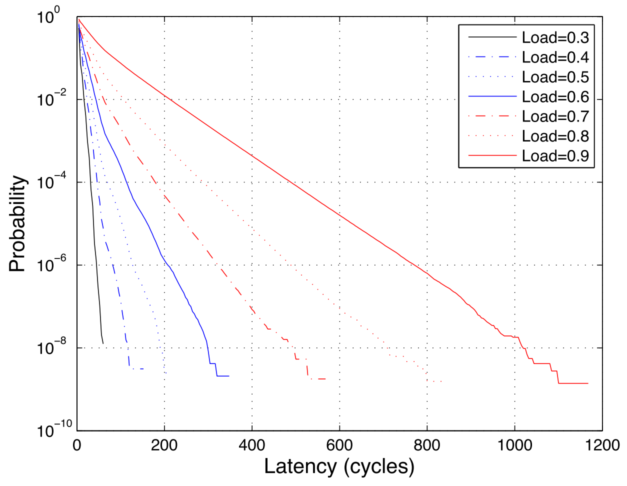

Fig. 3. Distribution of latency for a class with seven different input loads.

bank a Poisson distribution (the 'M' property) [1]. Second, a cell is mapped to multiple banks of a DRAM chip in an interleaved fashion. Since the cell size and mapping are chosen such that a row cycle time (tRC) of DRAM is smaller than a cell read or write time, the same bank can be accessed immediately after a cell read or write. This mapping makes cell reads or writes not depend on row hits or bank conflicts and guarantees the *deterministic* cell read and write time (the 'D' property). Fig. 3 shows the complementary cumulative distribution function (CCDF)[2] for the latency of cells scheduled by the QoS scheduler (modeled as a M/D/1 queue) with respect to different input load values. The distribution curve is unique for each load value which is the ratio of an incoming request rate over a service rate (an allocated memory bandwidth) and thus we can estimate the cell latency distribution when an incoming request rate and an allocated memory bandwidth are given. In other words, we can compute how much memory bandwidth is needed to obtain a certain distribution curve when an incoming request rate is given.

To facilitate understanding of how these distributions can be used, we provide the following example. In Fig. 3, the load value of 0.6 represents the case where an input load (i.e. an incoming request rate translated into an equivalent memory bandwidth) is equivalent to 60% of total available memory bandwidth. The complementary cumulative latency distribution curve for the input load of 0.6 represents the probability of cells whose latency is larger than the given latency value in x axis. These distribution curves can represent the latency requirements of a class because the latency requirement is given by the percentage of cells whose latency is larger than a certain threshold value. For instance, if a requirement for a class states that the probability of cells whose latency is greater than 200 cycles is less than 10^{-5}, it can be satisfied by lines with the load value less than 0.6. Based on this plot, we know in advance how much bandwidth should be allocated for each class to fulfill the latency requirement. If a class has the aforementioned latency

[2]Complementary cumulative distribution function is 1 - cumulative distribution function.

© IEICE 2014
DOI: 10.1587/elex.11.20130983
Received December 13, 2013
Accepted February 04, 2014
Publicized February 17, 2014
Copyedited March 10, 2014

requirement, i.e. probability of cell latencies greater than 200 cycles being less than 10^{-5}, allocating 100% of memory bandwidth can satisfy the latency requirement when the input load is 0.6. However, if the input load is only 0.3, allocating only 50% of memory bandwidth will produce the same latency distribution and satisfy the latency requirement.

4.2.2 Calculating bandwidth and weight for each class

From the latency distribution as shown in Fig. 3, the following proportional expression holds for class i:

$$L_{req}(i) : 100\% = L_{in}(i) : BW(i) \tag{1}$$

where $L_{req}(i)$, $L_{in}(i)$, and $BW(i)$ represent the latency requirement[3] and input load of class i and the bandwidth allocated to class i, respectively. Rearranging Eq. (1) gives

$$BW(i) = \frac{L_{in}(i)}{L_{req}(i)} \times 100\% \tag{2}$$

Note that the sum of all bandwidths should be less than 100%, namely

$$\sum_{i=0}^{n-1} BW(i) \leq 100\% \tag{3}$$

Otherwise, we should increase the overall memory bandwidth to obtain a stable system. For temporary overload cases, Eq. (3) may not be met.

Based on the bandwidth information, we can also determine the weight of each class for scheduling. Given n input classes, the following inequality should hold:

$$\frac{W(i)}{\sum_{i=0}^{n-1} W(i)} \times 100\% \geq BW(i) \tag{4}$$

where $W(i)$ represents the scheduling weight for class i. Assuming equality produces n equations

$$\frac{W(i+1)}{W(i)} = \frac{BW(i+1)}{BW(i)} \qquad \text{for } 0 \leq i \leq n-2 \tag{5}$$

$$\frac{W(0)}{W(n-1)} = \frac{BW(0)}{BW(n-1)} \qquad \text{for } i = n-1 \tag{6}$$

and solving these n equations gives the weight for each class[4]

For the cases where there are more traffics than the scheduler can handle, the proposed scheme assigns a high priority class the bandwidth needed to fulfill its requirement and assigns the remaining bandwidth to the rest.

4.2.3 Bitmap generation

Fig. 4 shows the proposed algorithm for generating a bitmap. For the sake

[3]Latency requirement is represented as a load value.

[4]For instance, if class 0, 1, and 2 require at least 10%, 30%, and 40% of total memory bandwidth respectively, solving the Eq. (5) and Eq. (6) produces weight 1, 3, and 4 respectively.

DOI: 10.1587/elex.11.20130983
Received December 13, 2013
Accepted February 04, 2014
Publicized February 17, 2014
Copyedited March 10, 2014

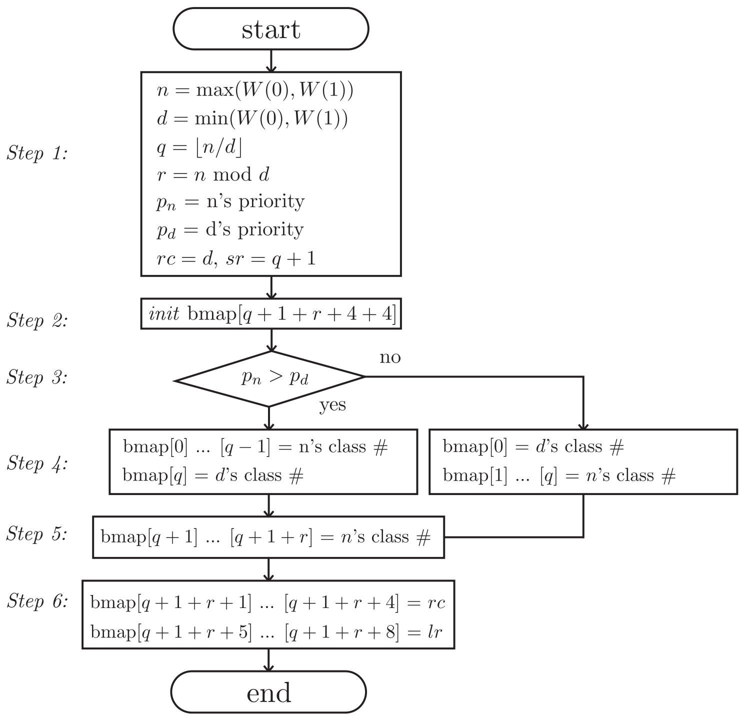
start
Step 1:
$n = \max(W(0), W(1))$
$d = \min(W(0), W(1))$
$q = \lfloor n/d \rfloor$
$r = n \bmod d$
p_n = n's priority
p_d = d's priority
$rc = d$, $sr = q + 1$
Step 2:
init bmap$[q + 1 + r + 4 + 4]$
Step 3:
$p_n > p_d$
no
yes
Step 4:
bmap[0] ... $[q - 1]$ = n's class #
bmap$[q]$ = d's class #
bmap[0] = d's class #
bmap[1] ... $[q]$ = n's class #
Step 5:
bmap$[q + 1]$ ... $[q + 1 + r]$ = n's class #
Step 6:
bmap$[q + 1 + r + 1]$ ... $[q + 1 + r + 4] = rc$
bmap$[q + 1 + r + 5]$ ... $[q + 1 + r + 8] = lr$
end

Fig. 4. Algorithm for generating the bitmap (n: numerator, d: denominator, q: quotient, r: remainder, p_n: priority of numerator, p_d: priority of denominator, rc: repetition count, lr: length of repetition part, bmap: bitmap).

of explanation, assume that there are only two input classes, namely class 0 (high priority) and class 1 (low priority). The inputs to this algorithm are the weight and priority of these two classes; the output is an (optimized) bitmap for scheduling classes 0 and 1. The bitmap generation can be explained with an example shown in Fig. 5 B. If the weight 7 and 3 are assigned for class 0 and 1, we want to generate a bitmap consisting of seven 0s and three 1s to allocate the correct bandwidth, i.e. 0 and 1 meaning scheduling class 0 and 1 respectively. To make the scheduler M/D/1, i.e. constant service rate, we would like to interleave the bit pattern as much as possible. Thus, the result bitmap becomes 0010010010. From the bit pattern, we can see that the bit pattern 001 is repeated three times and one additional 0 is at the end. The goal of the bitmap generation algorithm determines and stores only a repeated pattern, a repetition count, and a remainder portion from given weights instead of storing a un-optimized bitmap. The algorithm is described in Fig. 4 and each step is explained below.

- Step 1: We form a fraction in which the numerator, n, is set to the larger weight and the denominator, d, is set to the smaller weight. The integer division and modulo operation compute a quotient, q, and a remainder, r. In above example, the quotient determines the number of leading 0s in the repeated pattern. Adding 1 to leading 0s completes a repeated bit pattern whose length is $q + 1$. d becomes the repetition

© IEICE 2014
DOI: 10.1587/elex.11.20130983
Received December 13, 2013
Accepted February 04, 2014
Publicized February 17, 2014
Copyedited March 10, 2014

count. r is a number of 0s appended to the repeated bit pattern.

- Step 2: We allocate an array to store the bitmap. This array needs $q+1+r$ bits to store one repeated pattern and a remainder. In addition, the array needs four bits to store a repetition count, d, and another four bits to store the length of a repetition part, $q+1$[5].
- Step 3 and 4: If the numerator is the higher-priority class, then we set the leading q bits of the array to the class number of the numerator and set the $(q+1)th$ bit to the class number of the denominator. In above example, bmap[0] and bmap[1] are set to 0 (class 0) because q is two and bmap[2] is set to 1 (class 1). bmap[0] through bmap[2] become a repetition part. If the denominator is the higher-priority class, we then perform the opposite. The repetition part is always started with a high priority class number to give more priority to a high priority class in scheduling.
- Step 5: We complete the bitmap by inserting the class number of the numerator by as many times as the remainder value, r. In above example, bmap[3] is set to 0 because r is one.
- Step 6: Finally, fill two four bits allocated for a repetition count and a repetition part length. In the example, they are 3 and 3 respectively, which indicate the repetition part is repeated *three times* and the length of a repetition part is *three*.

When storing the bitmap in a lookup table, we store only a repetition part, a remainder, a repetition counter, and a repetition part length for a given pair of weights for two classes. The offline scheduler pre-compute the bitmaps for possible combinations of weights for two classes. These bitmaps are the output of a scheduler and stored in the second level lookup table in Fig. 5 A.

4.3 Hardware implementation: two-level lookup table

To get pre-computed scheduling bitmaps for given loads and latency requirements for two classes, we use a two-level lookup table shown in Fig. 5 A. The first level lookup is used to calculate the bandwidth[6] for a given input load, $L'_{in}(i)$, which is the number of cells arrived over a sampling period for class i. The second level lookup is used to look up the bitmap pattern for a given bandwidth (or weight) combination.

We quantize $L'_{in}(i)$ into q levels by comparing with pre-specified $q-1$ thresholds. For the binary class case, we need $(q-1)\times 2$ registers to store threshold values in the first level lookup table[7]. By comparing $L'_{in}(i)$

[5]Four bits for a repetition count and the length of a repetition part are empirically determined from experiments to optimize the cost and performance.

[6]The bandwidth is translated into an index to the second lookup table by the first lookup table.

[7]We vary the number of quantized levels from 5 to 30 and choose 15 levels for all experiments since it gives reasonable performance and a relatively small area cost.

© IEICE 2014
DOI: 10.1587/elex.11.20130983
Received December 13, 2013
Accepted February 04, 2014
Publicized February 17, 2014
Copyedited March 10, 2014

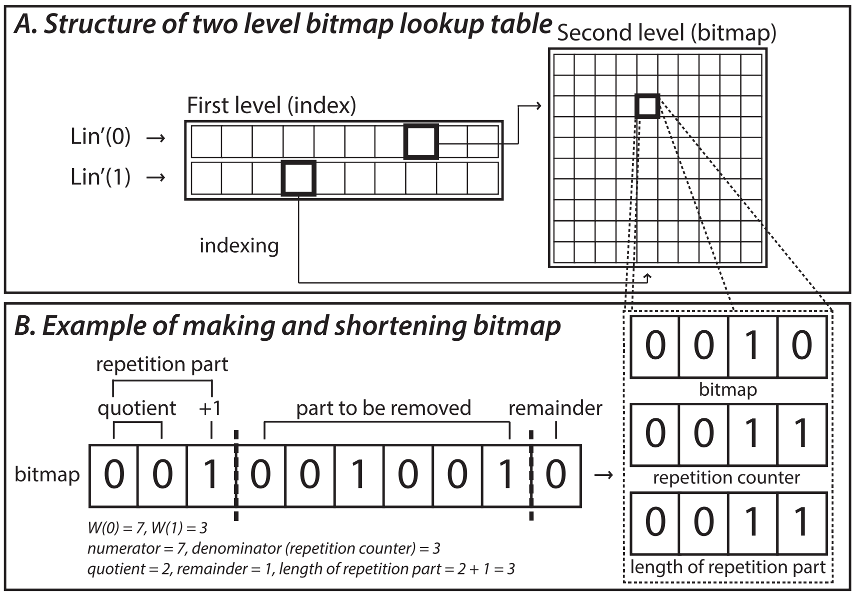
A. Structure of two level bitmap lookup table
Second level (bitmap)
First level (index)
Lin'(0) →
Lin'(1) →
indexing
B. Example of making and shortening bitmap
repetition part
quotient +1
part to be removed
remainder
bitmap
0 0 1 0 0 1 0 0 1 0
W(0) = 7, W(1) = 3
numerator = 7, denominator (repetition counter) = 3
quotient = 2, remainder = 1, length of repetition part = 2 + 1 = 3
0 0 1 0
bitmap
0 0 1 1
repetition counter
0 0 1 1
length of repetition part

Fig. 5. (a) Structure of a bitmap lookup table with two classes (0 and 1). (b) Example of optimizing a bitmap. Assume that $W(0) = 7$ and $W(1) = 3$. The repeated pattern, 001001, is removed from the original bitmap, 0010010010. The final bitmap, 0010, is stored into the second-level lookup table with the repetition count and the length of the repetition part.

with these registers in parallel and encoding the highest matching bit, we generate the index for the second level lookup table and access the table. The second-level lookup table contains a bitmap consisting of a repetition part, a remainder, a repetition counter, and a repetition part length, which is used by a QoS scheduler.

5 Experimental results

5.1 Simulation environments

An event driven simulator is developed to evaluate our proposed scheduler under various stressful scenarios. In all tests, we set the latency requirements for high and low priority classes to the distributions indicated by load of 0.7 and 0.8 in Fig. 3 respectively to create stressful scenarios. For SQMC, these requirements are translated into probability at 200 cycles in CCDF, which are 4.766×10^{-5} for HP class and 8.216×10^{-4} for LP class respectively.

In the next section, we present the results of tests that show the performance of our proposed method (BMS). The first test shows how BMS performs when the total effective load is periodically larger than total available bandwidth although its average value is smaller than the total available bandwidth. The second test rapidly switches the ratio between high and low priority traffic load although its total effective load is less than total available bandwidth.

© IEICE 2014
DOI: 10.1587/elex.11.20130983
Received December 13, 2013
Accepted February 04, 2014
Publicized February 17, 2014
Copyedited March 10, 2014

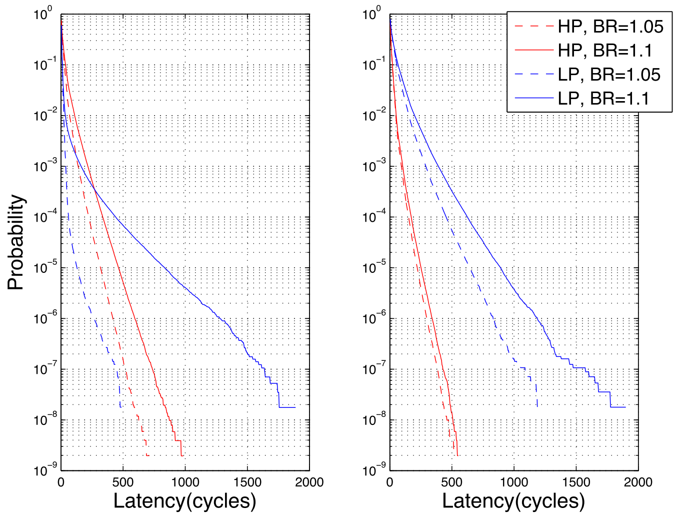

Fig. 6. CCDF of cell latency for SQMC (left) and BMS (right) when total effective load is periodically larger than total available bandwidth. (HP = high priority, LP = low priority, BR = overload ratio during burst periods, LR = latency requirement in load value)

5.2 Evaluation QoS scheduler

5.2.1 Total effective load is periodically larger than 100% of total available bandwidth

In this test, the total average effective input load, $\sum \frac{L_{in}(i)}{L_{req}(i)} \times 100\%$, is set to 99% of the level that a scheduler can handle. However, it increases periodically to larger than 100%. The input ratio between high and low priority is set to 9:1 and total input load increases by 5% (BR = 1.05) and 10% (BR = 1.1) during burst periods in two tests. Fig. 6 shows that both classes in SQMC increase the weight during overload periods and scheduling degenerates into round-robin, which violates the latency requirement for the HP class by a huge margin (1.7884×10^{-4} at BR = 1.05 and 0.0014 at BR = 1.1). Meanwhile, BMS adds more bandwidth to high priority by increasing weight and guarantees the latency of the higher priority class (1.8545×10^{-5} at BR = 1.05 and 4.2138×10^{-5} at BR = 1.1) when the scheduler cannot meet the requirement of all classes.

5.2.2 Input ratios for two classes are periodically changing while the total effective input load is close to 100%

In this experiment, we alternate the ratio between the input loads of class 0 and 1 while the total effective load is little less than 100% and compare the performance of BMS against SQMC. During the phase 0 and 1, the input ratio between class 0 and 1 is 9:1 and 2:8 respectively. The input rate change interval[8] is 1 MCycles and the dequeue rate change interval[9] is 1 KCycles.

[8]The input rate change interval is a parameter controlling how fast the input loads for different classes change.

[9]The dequeue rate change interval is a parameter that controls how fast the scheduler changes the schedule weight of each class, i.e. sampling window.

© IEICE 2014
DOI: 10.1587/elex.11.20130983
Received December 13, 2013
Accepted February 04, 2014
Publicized February 17, 2014
Copyedited March 10, 2014

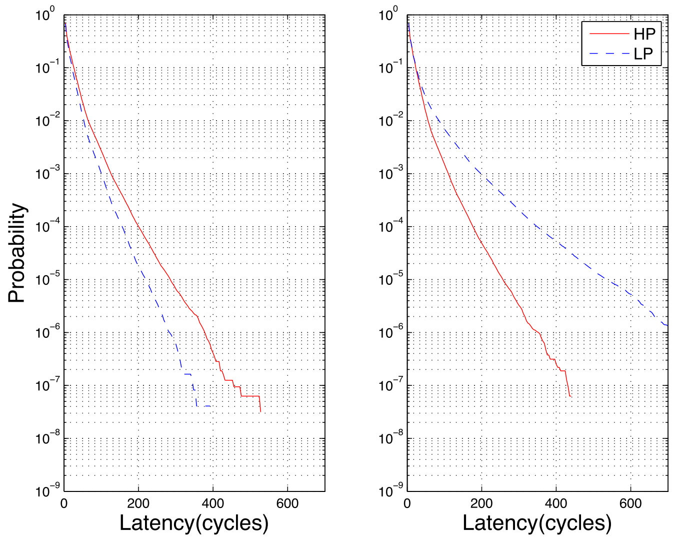

Fig. 7. CCDF of cell latency for SQMC (left) and BMS (right) when input ratio between high and low priority traffic is periodically changing.

Table I. Comparing probability at 200 cycles in CCDF for SQMC and BMS.

Pri	Lat. Req.	SQMC (Δ in %)	BMS (Δ in %)
HP	4.76×10^{-5}	10.1×10^{-5} (112%)	4.86×10^{-5} (2%)
LP	8.21×10^{-4}	1.79×10^{-5} (−97%)	9.63×10^{-4} (17.2%)

As shown in Fig. 7, SQMC shows that the high priority performance is severely degraded. This is because the ratio is changing rapidly and SQMC overreacts to errors from rapid input rate change. The weights for two classes continuously increase in SQMC and degenerate performance of high priority. The exact performance of the two methods for the second test are compared in Table I. The numbers inside parentheses are the percentage difference between the achieved probability and latency requirement in the second column. BMS shows much little difference from the latency requirement compared with SQMC.

6 Cost Analysis

If we quantize $L_{in}(i)$ into q levels, the first- and second-level lookup tables require $O(c(q-1))$ and $O(q^c)$ entries, where c is the number of classes. Typically, the maximum number of classes supported in the memory controller of routers is $2 \sim 3$ [7][10]. For the binary class case with 15 quantized ranges, 225 entries are sufficient for the second-level lookup table. If we as-

[10]Class of Services supported in the routers/switches can be larger, e.g. $2 \sim 7$ classes. However, the larger number of classes is only supported in the queue scheduler. The rest of the datapath is only supporting a small number of classes such as $2 \sim 3$ due to hardware complexity. The memory controller also supports only $2 \sim 3$ classes and provides some speedup over the queue scheduler so that the controller does not become a bottleneck of the performance.

© IEICE 2014
DOI: 10.1587/elex.11.20130983
Received December 13, 2013
Accepted February 04, 2014
Publicized February 17, 2014
Copyedited March 10, 2014

sume that the maximum length of a bitmap supports the weight ratio up tot $N : 1$, since the maximum length of the bitmap is less than the sum of the maximum quotient and the maximum remainder plus one, the number of bits required is $N + 1 + (N - 1) = 2N$. In other words, when the weight ratio is $N : 1$, we need allocate $O(2N)$ bits to the bitmap. In our design, we allocated 20 bits to the bitmap, considering up to $10 : 1$ ratio. Thus, for two classes, we use 2×14 (*thresholds*) $\times$ 16 bits for the first-level lookup and $225 \times (20$ (*quotient and remainder*) $+ 4$ (*repetition counter*) $+$ 4 (*repetition part length*)) bits for the second level lookup, which is $6.7K$ bits. For three classes, $169.4K$ bits are needed. Considering that tens of megabits of memory is used in the packet processor of a core router [7], this memory size and the cost of the comparators and a priority encoder are relatively small.

To prove the feasibility of the concept and confirm the cost and performance analysis, we design, verify, and synthesize the proposed QoS (class) scheduler. The design is implemented in Verilog and compiled via Synopsis design compiler for a Samsung 65 nm process. The target clock cycle time is set to 2 ns (500 MHz) to make synthesis challenging and area penalty conservative. The total number of flops reported is 6,812 which is in line with our estimation for two-class QoS scheduler[11]. Total combinational and non-combinational area are $8225.6\,\mu m^2$ and $11264.0\,\mu m^2$ respectively.

7 Conclusion

This paper proposes a novel packet memory scheduler that models the memory controller as an M/D/1 queue and creates a bitmap for scheduling. By adaptively varying the weight of each class, the proposed scheduler can process latency-sensitive high-priority packets, according to our experimental results. We expect that the proposed method would be able to guarantee the quality of service in modern internet environments in which routers are temporarily overloaded and input traffics abruptly changes.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No.2011-0023798 and 2012-008475).

© IEICE 2014
DOI: 10.1587/elex.11.20130983
Received December 13, 2013
Accepted February 04, 2014
Publicized February 17, 2014
Copyedited March 10, 2014

[11]The number of flops can be much smaller if we use a hard-macro memory block for the look-up table. We use flops because the hard-macro memory block is not available in the library.