Published in IET Communications
Received on 29th June 2009
Revised on 8th October 2009
doi: 10.1049/iet-com.2009.0404

ISSN 1751-8628

Dynamic queuing sharing mechanism for per-flow quality of service control

C. Hu[1,2] *Y. Tang*[1] *K. Chen*[3] *B. Liu*[1]
[1]*Department of Computer Science and Technology, Tsinghua University, China*
[2]*State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, China*
[3]*Department of Electrical Engineering and Computer Science, Northwestern University, USA*
E-mail: huc@ieee.org

Abstract: To achieve satisfying user experiences of diverse applications, quality of service (QoS) guaranteed mechanisms such as per-flow queuing are required in routers. However, deployment of per-flow queuing in high-speed routers is considered as a great challenge since its industrial brute-force implementation is not scalable with the increase of the number of flows. In this study, the authors propose a dynamic queue sharing (DQS) mechanism to enable scalable per-flow queuing. DQS keeps isolation of each concurrent active flow by sharing a small number of queues instead of maintaining a dedicated queue for each in-progress flow, which is novel compared to the existing methods. According to DQS, a physical queue is created and assigned to an active flow upon the arrival of its first packet, and is destroyed upon the departure of the last packet in the queue. The authors combine hash method with binary sorting tree to construct and manage the dynamic mapping between active flows and physical queues, which significantly reduces the number of required physical queues from millions to hundreds and makes per-flow queuing feasible for high-performance routers.

1 Introduction

Router-based per-flow quality of service (QoS) control mechanisms were proposed in the literatures to achieve advanced QoS guarantees [1, 2]. However, the traditional per-flow QoS control needs to keep state for each in-progress flow, and therefore it suffers from a great scalability problem because of the dramatic increase in the number of in-progress flows. Per-flow queuing is the classical per-flow QoS control mechanism in the router. Industrial per-flow implementation typically uses brute-force method to build a large number of physical queues and reserve a queue for every 'in-progress' flow. A flow is an individual internet protocol (IP) session specified by a five-tuple of source IP address, source port, destination IP address, destination port and protocol identifier. A flow is recognised as an in-progress one before its termination and an in-progress flow is generally considered to be terminated if an FIN packet is received for transmission control protocol (TCP) flow or it lasts for a pre-defined time interval. If the number of queues is larger than the number of in-progress flows, an isolated queue is assigned for each flow; otherwise several different flows would share a same queue and the QoS guarantee of each flow is violated. To avoid potential violation, at least 1 million queues need to be maintained for today's requirement with the brute-force method [3]. However, it is almost infeasible since the memory required to only keep the head and tail pointers of all the queues is about $2 \times 2^{20} \times \log_2 2^{20} = 40\,\mathrm{M}$ bits. Further considering the storage of packet data and the scheduling of 1 million queues, very large memory and powerful processing capability are needed for such brute-force methodology.

Related survey showed that the flow number could be up to millions in an hour nowadays [4, 5]. The trend that the increase of people's demand is much faster than that of computing hardware (e.g. processing speed, memory capacity) motivates us to seek a more scalable and smarter mechanism to support per-flow queuing in this paper. In fact, the time a packet stored in a router is typically in microsecond. We count the number of active flows (the flows having packets stored in the router) every 10 ms from different real traces, and observe that the number of active flows is only in hundreds as demonstrated in our previous work [6]. Similar results have also been reported in [7].

Based on this observation, we propose a dynamic queue sharing (DQS) mechanism to implement scalable per-flow queuing in routers. The idea is that we only assign queues for 'active flows' instead of in-progress flows in any snapshot while still keeping the per-flow queuing feature: packets from different flows are stored in different queues. An active flow is defined as a flow that has packets buffered in the router, and a silent flow is a flow that has no packet stored. DQS only creates and assigns a physical queue for an active flow and removes this queue when the flow turns to be silent. By setting only a limited number of queues and sharing them among simultaneous active flows, the number of required physical queues can be significantly reduced from millions to hundreds as we will show in Section 6.

The rest of the paper is organised as follows. Related work is briefly described in Section 2. Section 3 introduces the idea and work flow of DQS mechanism. Section 4 presents the organisations of active flow mapping (AFM) table. Section 5 shows one serial processing model and two parallel processing models on the AFM table for DQS mechanism. Section 6 performs the experiments and demonstrates the evaluation results. Finally in Section 7, we conclude the paper.

2 Related work

Packet buffering, acting as one of the basic functions in a router, is mainly responsible for holding arriving packets during the time of traffic congestion to smooth the burst of Internet traffic. In a packet buffering/queuing system, related studies fall into three categories: enqueue mechanism, dequeue mechanism and queue organisation mechanism.

Enqueue mechanism determines whether to accept the incoming packets or not and how to control the enqueue traffic rate, known as active queue management (AQM) [8] and admission control [9]. Random early detection (RED) [10] is the most prominent work in AQM and a large number of papers improve the stability [11], fairness [12] and self-adaptation [13] of RED in a single queue. AQM in a multi-queue/multi-class environment is also investigated in [14, 15]. In [16], per-flow admission control is proposed to allow QoS differentiation while maintaining the simple user–network interface of the best effort InterNetw. Proportional differentiated admission control is presented in [17].

Packet scheduling is the classical dequeue mechanism, which allocates the link capacity to different users and prioritises user traffic to meet various QoS requirements. In round-robin-based schedulers [18, 19], the server polls each queue in a cyclic order and serves a packet from any non-empty queue. Generalised processor sharing (GPS) [20] is an ideal scheduling policy in that it provides an exact max–min fair share allocation. However, GPS assumes that its scheduler is able to serve all backlogged sessions instantaneously and the capacity of the outgoing link can be infinitesimally split and allocated to these sessions. An important class of the so-called packet fair queuing algorithms can be defined in which the schedulers try to schedule the backlogged packets by approximating the GPS scheduler, such as worst-case weighted fair queuing [21], virtual clock [22], and self-clock fair queuing [23]. Adaptive packet scheduling methods, which guarantee bandwidth of the connection and optimise revenue of the network service provider, are proposed for both wireless and wired networks [24, 25]. In [26], the authors present the idea of utility-based scheduling disciplines for adaptive applications over the Internet.

Buffer/queue organisation is the basis of the queuing system. It decides how to implement queues in a buffer and how to associate flows with queues. This paper is in this category. The industrial implementation of per-flow queuing only provides a brute-force approach by means of a quite large number of physical queues to buffer the arrival flows [27]. Since the development of computing hardware is slower than the increase of link speed and flow numbers [4], brute-force methodology is not efficient and only guarantees a static-aggregating flow performance. The buffer management scheme to implement fair queuing in a gigabit router was proposed in [28]. It has recently been observed through trace studies and analytical evaluations that the number of active flows in packet scheduler is measured typically in hundreds even though there may be tens of thousands of flows in progress [7]. It indicates the feasibility of scalable per-flow queuing in [7], but it is still an unsettled problem that how to implement scalable per-flow queuing in practice. In our previous work [6], we have proposed a queue sharing mechanism, which significantly reduces the number of physical queues; however, its performance degrades greatly for burst traffic. In fact, the mechanism in [6] is the same as the serial processing model in this paper, and here we further propose two parallel processing models to improve performance under the burst traffic.

3 Work flow of DQS

For easy reference, we summarise the notations used in Table 1. Suppose that the flow universe is $F = \{f_i\}$, the set of active flow is $A = \{a_j\} A \subseteq F$, and the set of physical queue is $Q = \{q_k\}$. We need to keep mapping between active flows and physical queues denoted by $a_j \rightarrow q_k$, which is called the AFM. An AFM table is introduced to keep the mapping between active flows and physical queues, in which each entry contains two data fields: the identifier of the (active) flow and the identifier of its corresponding physical queue. When a flow's state changes from active to silent, its mapping entry is deleted from the AFM table and the corresponding physical queue is withdrawn. When

Table 1 Notations used in the paper

Notations	Description
$F = \{f_i\}$	flow universe
$A = \{a_j\}$	the set of active flows
$Q = \{q_k\}$	the set of physical queues
$S = \{s_k\}$	the set of sub-tables (slots)
M	the number of sub-tables (slots)
B	the number of bits to identify a flow
N	the number of physical queues
I	the number of active flows
W	the memory required for the active mapping scheme

the flow changes its state from silent to active, a physical queue is allocated to this flow and its mapping entry is created in the AFM table. The following packets from this flow will be buffered in this dedicated queue correspondingly during the active period of the flow. The work flows of the processing for packet arrival and departure are illustrated in Fig. 1. The flow identifier is first extracted from the packet and then forwarded to an AFM module. When a packet arrives from f_i, the module first checks whether the flow f_i is in the active list A or not by looking for f_i in the flow identifier field of AFM table entries. If a hitting entry is found, the data q_k in the queue identifier field will be returned and the packet is pushed into the corresponding q_k; otherwise, a new queue q_k is assigned and an entry $f_i \rightarrow q_k$ is added. Once the physical queue becomes empty, it is withdrawn immediately and its mapping entry $f_i \rightarrow q_k$ is removed from AFM table.

Instead of maintaining the states of all the flows, DQS only needs to manage a small number of physical queues, and a mapping table between active flows and physical queues. As a result, the scheduler is only aware of the existence of physical queues rather than all the in-progress flows, without sacrificing the per-flow management feature. Consequently, the states in a router are reduced to hundreds from millions at a cost of a delicate maintenance of queues and operations on AFM table. The AFM scheme will be discussed in the next section.

4 AFM scheme

The scheme presented in this section maintains the mapping between flows and queues and helps a newly incoming packet to find a proper queue to be pushed into. The set of active flow A is a subset of flow universe F and changes from time to time. At time t_1 and time t_2, it may be the same in the quantity of flows, but the individual flows may have changed a lot. The mapping scheme maintains a one-to-one mapping between active flows A and physical queues Q.

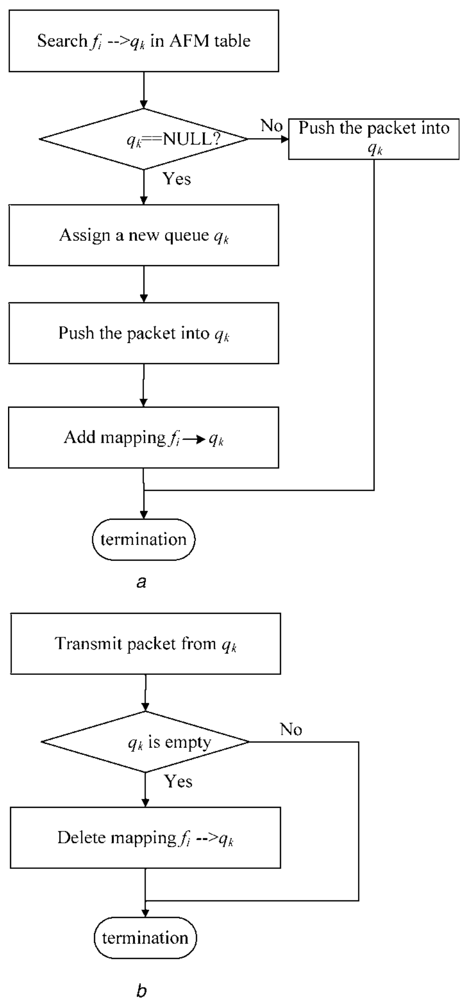
Search f_i -->q_k in AFM table
q_k==NULL?
No
Push the packet into q_k
Yes
Assign a new queue q_k
Push the packet into q_k
Add mapping $f_i \rightarrow q_k$
termination
a
Transmit packet from q_k
q_k is empty
No
Yes
Delete mapping f_i -->q_k
termination
b

Figure 1 *Workflow of DQS*
a Processing when a packet arrives
b Processing when a packet departs

A queue is created on demand when a flow starts and is removed when it terminates (more details are described in Section 4.3), thus the numbers of elements in A and Q are the same, but this number is much smaller than the elements number in F. The mapping scheme requires a search operation, which helps a packet (from an existing active flow) find the proper queue to be pushed into. In addition, the mapping scheme maintains insertion operations to create new entries for new flows and holds deletion operations when specific physical queues are removed from the AFM table.

As we mentioned before, the five-tuple in the packet header is employed to classify the flows. Since the five-tuple in the packet header has 13 bytes (for IPv4), the flow universe contains 2^{104} elements totally. The number of active flows in the buffer is very small relative to F, but it is not an easy job to frequently search, insert and delete a mapping entry in an extremely short interval, especially when A changes frequently. We conquer the problem in two steps. First, we use hashing to divide the whole AFM table into a number of smaller sub-tables, and in this way, the operations on these small AFM sub-tables can be much faster. Second, we employ binary sorting tree (BST) to organise the divided sub-tables, where search, insertion and deletion can be easily and quickly performed.

4.1 AFM table splitting

Hashing is utilised to divide the whole AFM table into smaller ones as indicated in Fig. 2. An array (direct-address table) is introduced and denoted by $S = \{s_k, k = 1, \dots, M\}$, in which each slot s_k in the array points to a sub-table. Flows with the same hash value $h(f_i) = k$ will be pointed to the same slot s_k to construct the AFM sub-table. The shade area of active flow A in Fig. 2 will move from time to time within the range of F, so the pointers to the sub-tables (S) also change dynamically.

We have examined a number of hash functions to find an even one so that approximately same amount of flows can be hashed to each slot. As our previous work [6] indicated, cryptographic hash functions such as MD5 and SHA-1 distribute flows evenly to slots. In a real implementation of DQS using FPGA (field-programmable gate array), we use another efficient hardware hash function: H_3 function [29]. The hash calculation can be finished within one clock cycle (the FPGA used in our prototype is ALTERA Stratix EPS80 and the frequency is 166 MHz) since the H_3 only requires AND operations and XOR operations. We evaluate AFM splitting effect using H_3 function with three real traces: trace 'CERNET', collected from CERNET (China Education & Research Network) [30]; trace 'NLANR1' and trace 'NLANR2' collected from [31]. We observe that the variation of the flow number in slots is slight. Fig. 3 shows the number of flows pointed to each slot after hashing with 64 slots and 256 slots. Note that we do not consider the departure of flows in this experiment, that is all the flows in each trace (each trace lasts 10 min) are considered to be active flows. If we take into account the scheduling (We adopt DRR [18] as the scheduling algorithm and the scheduling rate is set so as to make the traffic load to be 0.95.) and only count the concurrent active flows, the number of active flows hashed to each slot

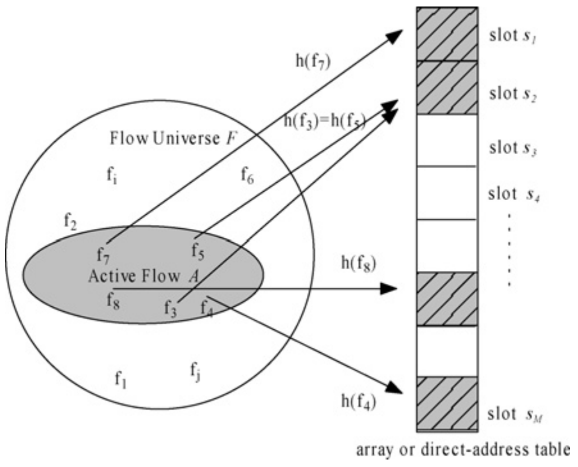

Figure 2 *Divide AFM table using hashing*

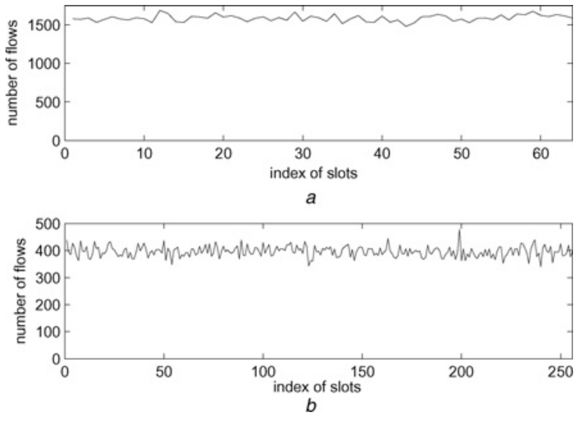

Figure 3 *Number of flows hashed to each slot without leaving flows*

a There are 64 slots in total
b There are 256 slots in total

(i.e. the length of sub-table in each slot) is quite small. Table 2 gives a summary of the experimental results, where $P(i)$ means the probability that more than i flows hashed to a same slot and α is defined as the ratio of the average active flow number to the total slot number. Sub-table length is the number of flows hashed to that sub-table and

Table 2 Active flows in each table slot using H_3 hashing considering scheduling

		CERNET	NLANR1	NLANR2
64	mean sub-table length	2.65	2.12	2.91
Slots	worst sub-table length	15	13	16
	α	2.28	1.49	2.70
	$P(1)$	0.531	0.346	0.693
	$P(2)$	0.409	0.173	0.419
256	mean sub-table length	1.34	1.24	1.42
slots	worst sub-table length	8	7	9
	α	0.57	0.37	0.68
	$P(1)$	0.118	0.049	0.149
	$P(2)$	0.024	0.007	0.03
1024	mean sub-table length	1.09	1.03	1.12
slots	worst sub-table length	5	5	7
	α	0.14	0.09	0.17
	$P(1)$	0.007	0.003	0.016

the average sub-table length in Table 2 only considers the sub-tables with at least one flow mapping, which should be larger than one as a result. By introducing more slots, the average and the worst case sub-table length decrease, as well as $P(1)$ and $P(2)$. The results demonstrate that the sub-table length is not very large and there is small probability for queue length to be larger than 3 even only 64 slots are employed. In general, the larger the number of slots, less flows hashed to a slot.

4.2 AFM sub-table organisation

We introduce the BST to organise all the flows hashed to the same slot. Linked list can also be used here, and BST is employed to improve the operation performance on the slot whose length is large. The detailed comparisons of using linked list and BST are described in [6]. Owing to the page limit, only the organisation of BST is presented in this paper. The pointer in slot s_k now points to the root of a constructed BST containing the flows hashed to s_k, or to NULL (no flow hashed to it). For any node y in a BST, the key value is the flow identifier number, which follows the BST property: keys in the left subtree of y are smaller than the key in node y, and the keys in the right subtree of y are larger than the key in node y. The operations of insertion and deletion of a flow cause the AFM sub-tables to be changed, but the BST property in each sub-table always holds.

(1) *Search:* The search operation begins at the root and traces a path downwards in the tree. For each node it encounters, it compares the identifier number of the searched flow with the key value stored in that node. If they are equal, the search terminates. If the identifier number of the searched flow is smaller than the key value in the node, the search continues in the left subtree of the node, since the search key could not be stored in the right subtree as indicated by the BST property. Symmetrically, if the identifier number of the searched flow is larger than the key value of the node, the search continues in the right subtree.

Note that the nodes encountered during the search recursion form a path downwards from the root, and thus the running time of search operation is $O(h)$, where h is the height of the tree. Therefore, in this case, the average searching time can be estimated by the following lemma and theorem.

Lemma 1: The average search length SL_k of a randomly built BST with k nodes is

$$\mathrm{SL}_k = \begin{cases} 0 & k = 0 \\ 1 & k = 1 \\ \dfrac{2}{k} - 1 + 2\left(1 + \dfrac{1}{k}\right)\displaystyle\sum_{i=2}^{k}\frac{1}{i} & k \geq 2 \end{cases} \tag{1}$$

Proof: Please see Appendix.

Denote P_k as the probability that k flows are hashed to a same slot, and it can be presented as

$$P_k = \binom{l}{k}\left(\frac{1}{M}\right)^k\left(1 - \frac{1}{M}\right)^{l-k} \tag{2}$$

Theorem 1: When the AFM sub-tables are organised by BSTs, and P_k is defined in (2), under the assumption of simple uniform hashing, the expected time of a successful search is

$$l\left(\frac{1}{M}\right)\left(1 - \frac{1}{M}\right)^{l-1} + \sum_{k=2}^{l} P_k\left[\frac{2}{k} - 1 + 2\left(1 + \frac{1}{k}\right)\sum_{i=2}^{k}\frac{1}{i}\right] \tag{3}$$

Proof: Please see Appendix.

(2) *Insertion:* Insertion only occurs after an unsuccessful search for a new active flow. When a new active flow comes, we first compare the flow identifier number with the key in the root. If the identifier number is smaller, we continue to trace the left subtree, and otherwise we continue to trace the right subtree. This procedure continues until it reaches the bottom of the tree, then a new node is created in this position with the flow identifier as the insertion key value. Consequently, the inserted node must be a leaf node.

Theorem 2: When the AFM sub-tables are organised by BSTs, under the assumption of simple uniform hashing, the expected time to identify a new active flow, is bounded by

$$\sum_{k=1}^{l}\binom{l}{k}\left(\frac{1}{M}\right)^k\left(1 - \frac{1}{M}\right)^{l-k}\left[\log_2\binom{k+3}{3} - 1\right] \tag{4}$$

Proof: Please see Appendix.

In fact, the insertion repeats the searching procedure to find a proper position to insert the node. Consequently, we can combine search operation with insert operation to solve our specific problem of flow mapping. If the incoming flow is not found in the BST, a new node is created at the position where the search operation terminates; otherwise, the searching result is returned.

(3) *Deletion:* Each node contains a pointer to its parent. Suppose that the node to be deleted is x whose parent is p. The deletion of the node x is also not complicated and we discuss it in three cases (without loss of generality, we assume x is the left child of p):

- If the node x is a leaf node, that is, x has no children, we just need to change the pointer of its parent p, which originally points to x, to NULL after deletion. It will be achieved in $O(1)$ time.

- If the node x has only a single left (or right) child, we delete x by making p directly point to the left (or right) child of x. It will operate in $O(1)$ time.

- If the node x has two children, we first make x's right child as the right child of the node with largest value in its left subtree (can be spliced out by in-order traversing its left subtree), and then make x's left child as the child of p. The running time is $O(h)$ in this case, where h is the height of the BST.

4.3 Management of physical queues

An intuitive and simple way to organise the queues is to permanently divide the buffer into N queues. Since in this situation, the states of all N queues need to be stored all the time. Obviously, it will be inefficient for large N; however, if N is small, there will be a risk of blocking packets when all the queues are occupied during busy periods. Static dividing of queues may be a tradeoff between implementation and performance. Unfortunately, many applications cannot tolerate potential performance degradation. We develop a mechanism to dynamically create and release queues. For management efficiency, we divide the storage space into blocks. All the blocks are of the same size, saying 64 bytes. A packet is segmented into a number of fixed-length data units (DU) (a data unit and a block are of same size), and blocks are assigned to buffer the packets. Blocks not assigned are called free blocks. A stack is utilised to manage the free blocks, which stores the pointers to the free blocks. Each physical queue is organised as a linked list to store all the DUs for the same flow. When a new packet arrives, one or more blocks are allocated depending on the packet size for the corresponding queue; and once being forwarded, all the blocks occupied by the packet will be withdrawn and pushed into the free block stack. Corresponding head and tail pointers of each queue, as well as the free block stack, can be stored in a small on-chip or an external SRAM. Therefore with such a buffer organisation a physical queue can be dynamically created when a new flow enters the buffer and dynamically released when all the packets in it have been scheduled out, that is, the queue becomes empty.

5 Processing models

In this section, we propose three processing models of DQS on AFM table: single-mapping-process, single-task-queue (SPSQ), multiple-mapping-process, single-task-queue (MPSQ) and multiple-mapping-process, multiple-task-queue (MPMQ).

In SPSQ, there is only one process that operates on all the AFM sub-tables and a task queue that buffers the entire backlog mapping requests. It is a serial processing model, where the flow ID is first hashed to an AFM sub-table and a mapping process will then search the sub-table. Other mapping requests of the following packets coming during the processing period will wait in a single task queue after the hash operation. The architecture is as shown in Fig. 4*a*. This processing model is the same as the mechanism in our previous work [6]. Since the items kept in different AFM sub-tables are independent, the operations on different AFM sub-tables can be performed in parallel and we further propose two parallel operation models, MPSQ and MPMQ on AFM table.

In the MPSQ model, it maintains one mapping process for each AFM sub-table that is not empty and a single task queue for all the backlog mapping requests as depicted in Fig. 4*b*. An incoming mapping request is first forwarded to the hash function and is marked with an identifier to indicate which sub-table as well as the corresponding mapping process it is hashed to. After that, it is kept in the task queue like what SPSQ does. The request in the head of the task queue can be directly sent to a mapping process corresponding to the AFM sub-table it hashed to if this mapping process is idle. Only when the consecutive requests are hashed to a same sub-table, the task queue will grow. Because multiple mapping processes exist in the MPSQ, the mapping results will leave the AFM table in a different sequence from the original one as the mapping request comes. The function of the ordering logic is to

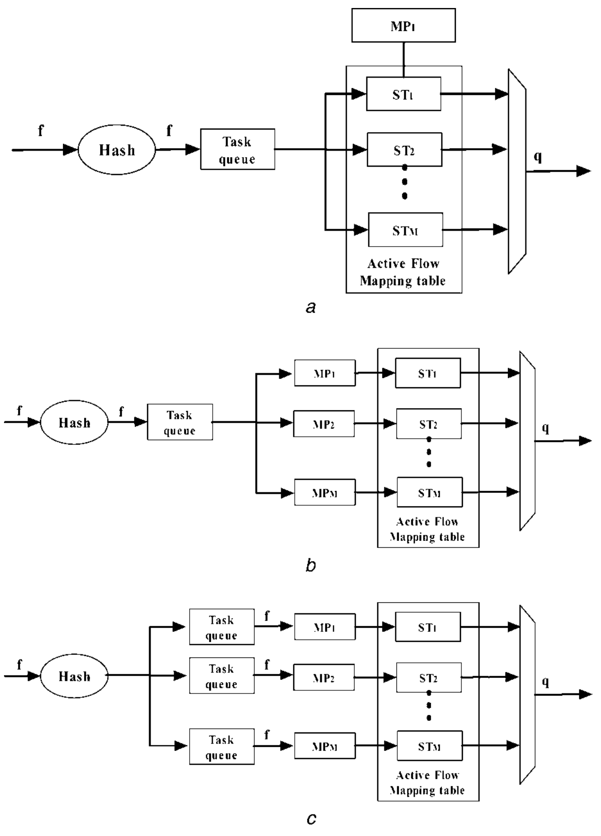

Figure 4 *Processing model (MP stands for mapping process and ST stands for sub-table in the figure)*
a Logical architecture of SPSQ
b Logical architecture of MPSQ
c Logical architecture of MPMQ

insure that the results will be returned in the same order as that of the input side. An architecture based on tag-attaching is used in the ordering logic. When an incoming mapping request is distributed to the proper AFM sub-table, a tag (i.e. sequence number) will be attached to it. Then at the output side, the ordering logic uses the tags to reorder the returning sequence. The allocation of new queues to each sub-table is in serial to avoid possible conflict.

MPSQ improves the performance of SPSQ; however, MPSQ will suffer 'head of line (HOL) block' problem. Suppose there are three consecutive mapping requests. First two are hashed to ST_1 and the third one is hashed to ST_2. Since the mapping process related to ST_1 is occupied by the first request, the second request has to wait in the task queue and the third one will also be blocked even the mapping process corresponding to ST_2 is idle. To solve this problem, we extend the single task queue of MPSQ into M task queues, where M is the number of sub-tables. That is the MPMQ structure as shown in Fig. 4*c*. There is one mapping process and one small task queue for each AFM sub-table that is not empty. Since each AFM sub-table has its own task queue, the HOL block problem does not exist. Similar to MPSQ model, there is also a reordering logic in MPMQ model to keep the sequence of output as input requests. The same as MPSQ, in MPMQ, the allocation of new queues to each sub-table is in serial to avoid possible conflict.

6 Evaluations

We utilise the aforementioned three traces to do the evaluations, namely CERNET trace, NLANR1 trace and NLANR2 trace (on OC-192 link). In the burst, the peak traffic rates (We define traffic rate as follows: suppose the packets arrive at time $t_0, t_1, t_2, \ldots, t_n$ and the packet length is $z_0, z_1, z_2, \ldots, z_n$. Then the traffic rate at time t_i is $z_i/(t_{i+1} - t_i)$. The peak traffic rates are the largest traffic rate.) of these three traces are 10, 2.5 and 1 Gb/s, respectively. In all the simulations illustrated in this section, output bandwidth is shared among all the active physical queues in a DRR manner [18].

The number of required physical queues is first evaluated. From Table 3, we observe that the maximum (and average) number of physical queues for each trace is only in the order of hundreds. This fact accords with the expectation that the queue number required is only in hundreds. Note that, in the experiments where we configure a lighter work load by increasing the output capacity, the number of physical queues is even smaller. Although the average traffic speed in CERNET trace is the smallest, the number of flows in this trace is larger than the number of flows in NLANR1. For this reason, the physical queues needed for CERNET trace are more than that for NLANR1.

Table 3 Number of required physical queues

Traces	CERNET	NLANR1	NLANR2
average trace speed (Mb/s)	278.8	447.4	662.6
output capacity (Mb/s)	293.5	470.9	697.4
max. queue number	352	239	385
average queue number	146	95	173

Next, we check the operation time including searching time, updating time and overall operation time under SPSQ processing model. Fig. 5 shows the searching time for an existing flow and Fig. 6 depicts the time to identify a newly arrived active flow and update the AFM table. The results in these two figures do not consider the computation time of hash function. The theoretic bound stated in Theorems 1 and 2 are acceptable estimations compared to the experimental results, and the margins between the theoretic values and the experimental results are partially caused by that the hashing is not so uniform among different slots/AFM sub-tables. With the increase of the number of AFM sub-tables, both the searching time and updating time decrease. This trend is just as expected, since the increase of M leads to the decrease of the scale of AFM sub-tables. If $M \geq 256$, in all the traces, the average time to search for an existing flow is less than 1.5 operation cycles and the average time to identify a new active flow is less than one operation cycle. For the sake of clearly distinguishing the lines in the figure, the results for NLANR2 and CERNET trace are not drawn in Figs. 5 and 6, which are similar. Taking into account the time to compute the hash function, we investigate the average clock cycles needed to return a queue q_k for an incoming packet no matter whether it belongs to a newly incoming flow or an existing active one. Suppose the hash computation needs one clock cycle, and each comparison on one node of BST also needs one clock cycle.

Fig. 7 illustrates the experimental results for the average overall operation time on CERNET trace, NLANR1 trace and NLANR2 trace. This figure clearly demonstrates two

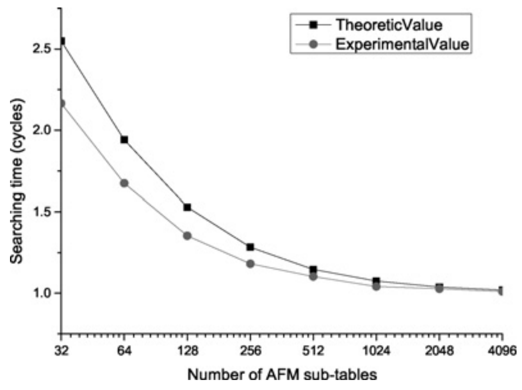

Figure 5 *Time cycles needed for searching an existing active flow on trace NLANR1*

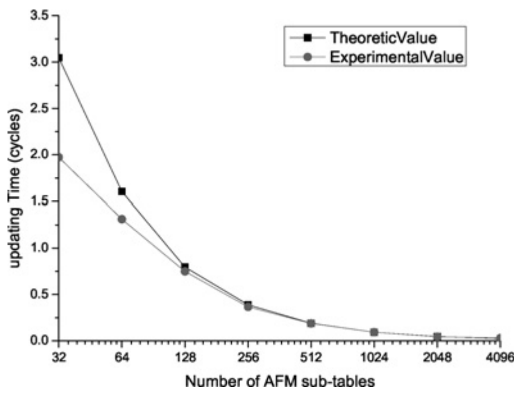

Figure 6 *Time cycles needed for updating the AFM table on the arrival of a newly incoming flow on trace NLANR1*

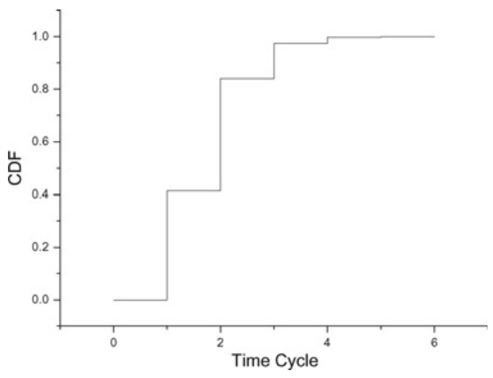

Figure 8 *Cumulative distribution function (CDF) of operation clock cycles*

facts: (i) If the number of slots is well set (for example, no less than 256), a packet can find a queue to be pushed into after about two cycles on average, in all the three traces. (ii) The operation time needed is a monotone decreasing function of slots number; however, it has a diminishing gain. Consequently, blindly increasing the number of slots will lead to ineffectiveness in the sense of incurring disproportional penalty in implementation cost vis-á-vis the gain in decreasing search length. Considering the overhead of increasing slots, we believe that it should be a good tradeoff containing 256 slots. In the worst case when many flows are occasionally hashed to one slot, the operation clock cycles needed will be more than the average results shown in Fig. 7. We investigate the effect of the worst case in Fig. 8. This figure shows the CDF of operation clock cycles in NLANR2 trace, when the flows are hashed to 256 slots. We observe that about 40% of the packets can find their corresponding queues after only one clock cycle, and about 84% of the packets can find their corresponding queues in no more than two clock cycles. Only about 0.02% of the packets may encounter worst case and require at most six operation cycles. The results for CERNET trace and NLANR1 trace are much better. Therefore for the worst case concern: (i) the worst case may occur and about six operation cycles are required in this case; (ii) the chance of meeting the worst case is very small (less than 0.02% in this case).

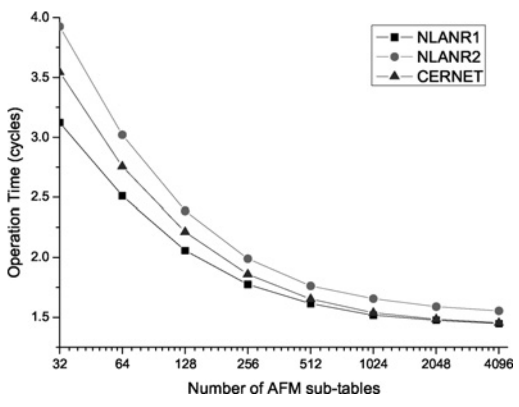

Figure 7 *Average overall operation time: clock cycles needed to return a queue identifier q_k for an incoming packet*

A state-of-art high-performance router usually equips combined input and output queue switch architecture and the switch fabric is required to provide a minimum speedup of $2 - 1/X$ in order to provide 100% throughput in such an architecture, where X is the number of input/output ports [32]. In other words, the enqueue rate of the output queue is $2 - 1/X$ times faster than the dequeue rate. Define speedup as the ratio between enqueue rate and dequeue rate. The performance of SPSQ will degrade greatly when the speedup is larger than one. The results for trace NLANR1 are shown in Fig. 9 for illustration. For this reason, we propose two parallel processing models to improve the performance when the burst speedup is larger than one. Fig. 10 depicts the relationship between normalised throughput and speedup under different processing models when the number of AFM sub-tables is 2^6 on trace NLANR1. Here, by normalised throughput we mean the AFM mapping output rate against the packet incoming rate. The normalised throughput is always bounded by 1 because the mapping output rate can never exceed the incoming rate. As shown in the figure, the normalised throughput decreases with the increase of speedup for all the three processing models. The normalised throughput of MPSQ is only slightly better than the one of SPSQ when speedup is larger than 3 since the probability of HOL block is quite tremendous for large speedup, but MPMQ is always about five times better than SPSQ.

The buffer sizes for the task queue under the three processing models are demonstrated in Fig. 11 when the number of AFM sub-table is 2^6 on trace NLANR1. In this paper, the buffer size is measured in K bytes. The buffer size for MPSQ is smaller than the one of SPSQ when speedup is less than 3, and they become similar with continuous increase of speedup. We calculate the total memory of MPMQ to be the product of the number of

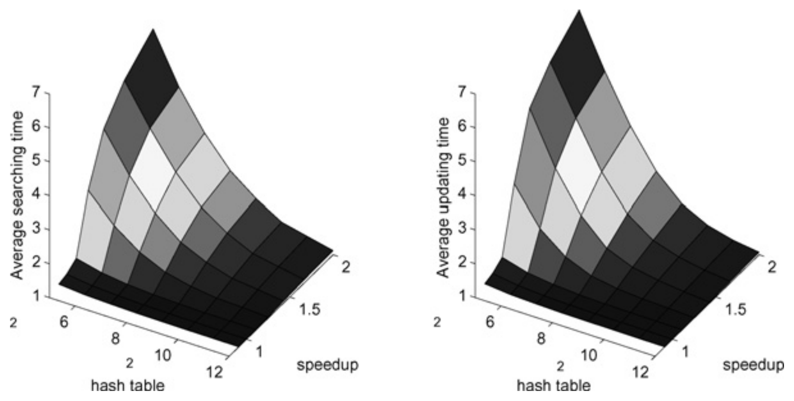

Figure 9 *Searching time (left) and updating time (right) when the speedup is larger than one on trace NLANR1*

sub-tables and the size of the largest task queues. The memory size of MPMQ increases with the growth of the speedup and it is the least when the speedup is no larger than 4. Since MPSQ and MPMQ provide larger throughput than SPSQ, the backlogged packets in MPSQ or in each MPMQ task queue should be smaller than SPSQ. MPMQ under large speedup requires largest total buffer since it has to multiply the large number of queues. In order to see the performance under tough situation, we fix the speedup to 8 and check the performance with the increase of the number of AFM sub-tables and the results on NLANR1 are indicated in Figs. 12 and 13. With the increase of the number of sub-tables, the normalised throughput increases while the buffer size decreases. This is because when the size of sub-tables becomes larger, the possibility of hash collision on successive packets becomes less. Again, the throughput performance of MPMQ is the best compared with SPSQ and MPSQ under this experiment setting. Although the backlog (largest buffer size) in each task queue of MPMQ model is less than 3KB, the total required memory size could be 172–550 KB.

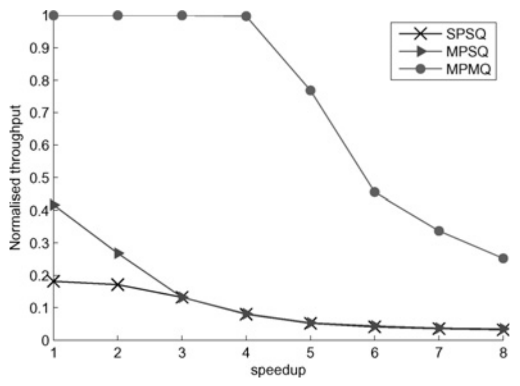

Figure 10 *Normalised throughput against speedup when the number of AFM sub-table is 2^6*

The memory required for the mapping scheme is also very small. As shown in Table 1, we denote the number of slots as M, the number of active flows as l, the number of physical queues as N and the number of bits to identify a flow as B. Then, the memory requirement for the mapping scheme W is

$$W = M + l(3\log_2 l + \log_2 N + B) \quad (5)$$

For example, given 256 slots, 256 physical queues, $l = 256$ and $B = 32$, the memory required is only 16 K bits for the AFM sub-tables. Obviously, it is easy to implement the memory on an embedded on-chip SRAM, leading to a great reduction on PCB area and wiring delay.

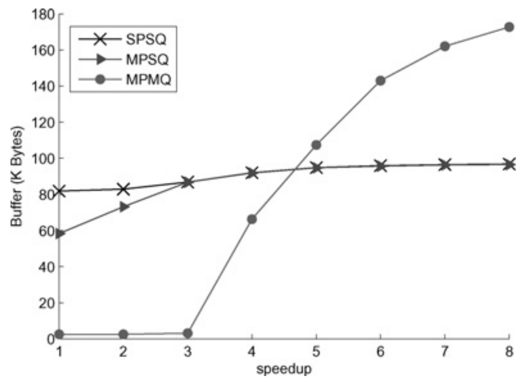

Figure 11 *Buffer size for task queue(s) against speedup when the number of AFM sub-table is 2^6*

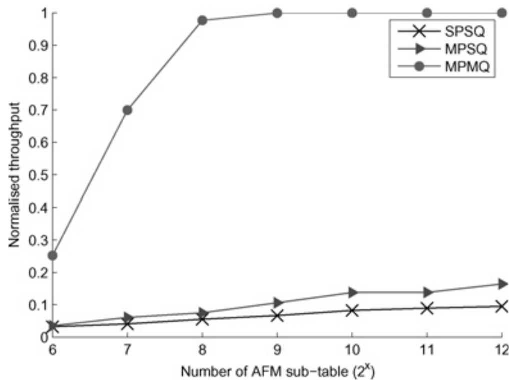

Figure 12 *Normalised throughput against the number of AFM sub-tables when speedup is 8*

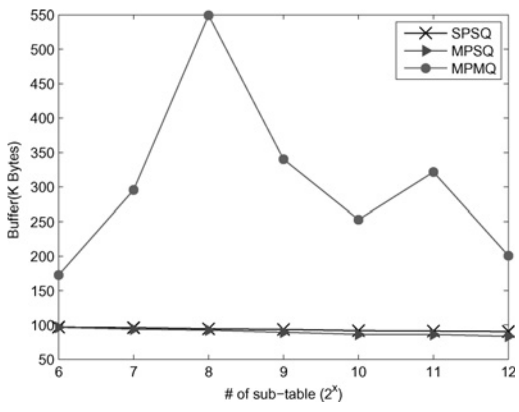

Figure 13 *Buffer size for task queue(s) against the number of AFM sub-tables when speedup is 8*

7 Conclusions

Per-flow queuing mechanism suffers a great scalability problem with the dramatic increase in the number of in-progress flows. In this paper, we first confirm the measurement result that the number of concurrent active flows in the routers is typically in hundreds even though there may be tens of thousands of flows in-progress. Based on this observation, we have proposed a DQS mechanism to implement scalable per-flow queuing in high-performance routers.

Compared with industrial implementation of per-flow queuing, the main benefit achieved by DQS is a great reduction of the number of the required physical queues from millions to hundreds at the expense of organisation and operation on an AFM table. When the number of slots of AFM is 256, the operation delay for a packet to find a queue to be pushed into is about two cycles on average under a serial processing model for normal traffic condition. Compared to serial processing model, parallel processing models improve the throughput performance by about five times even under congested conditions. With 256 slots, the system only costs 16 K bits memory, which can be implemented in an on-chip memory. In addition, the idea of DQS can be extended to other per-flow control mechanism which is also our future work.

8 Acknowledgments

This work is supported by NSFC (60625201, 60873250, 60903182), Specialized Research Fund for the Doctoral Program of Higher Education of China (20060003058), 973 project (2007CB310701) and open project of state key laboratory of networking and switching technology (SKLNST-2008-1-05). Part of the work is presented at ‘Per-flow queuing by dynamic queue sharing’, IEEE INFOCOM 2007.

9 References

[1] SIEW C.K., FENG G., LONG F., ER M.H.: ‘Congestion control based on flow-state-dependent dynamic priority scheduling’, *IEE Proc. Commun.*, 2005, **152**, (5), pp. 548–558

[2] BRADEN R., CLARK D., SHENKER S.: ‘Integrated services in the internet architecture: an overview’, *RFC1633*, 1994

[3] http://www.lightreading.com/document.asp?site=lightreading&doc_id=22628&page_number=8, accessed Augest 2006

[4] ESTAN C., VARGHESE G.: ‘New directions in traffic measurement and accounting’. Proc. ACM SIGCOMM, Pittsburgh, US, August 2002, pp. 323–336

[5] FRALEIGH C., MOON S., LYLES B., *ET AL.*: ‘Packet-level traffic measurements from the sprint IP backbone’, *IEEE Netw.*, 2003, **17**, (6), pp. 6–16

[6] HU C., CHEN X., TANG Y., LIU B.: ‘Per-flow queueing by dynamic queue sharing’. Proc. INFOCOM, Anchorage, US, May 2007, pp. 1613–1621

[7] KORTEBI A., MUSCARIELLO L., OUESLATI S., ROBERTS J.: ‘Evaluating the number of active flows in a scheduler realizing fair statistical bandwidth sharing’. Proc. ACM SIGMETRICS, Banff, Canada, August 2005, pp. 217–228

[8] BRADEN B., CLARK D., CROWCROFT J., *ET AL.*: ‘Recommendations on queue management and congestion and congestion avoidance in the internet’, *RFC2309*, 1998

[9] MASSOULI L., ROBERTS J.: ‘Arguments in favour of admission control for tcp flows’. Proc. 16th Int. Teletraffic Congress, Edinburgh, Scotland, June 1999, pp. 33–44

[10] FLOYD S., JACOBSON V.: ‘Random early detection gateways for congestion avoidance’, *IEEE/ACM Trans. Netw.*, 1993, **1**, (4), pp. 397–413

[11] OTT T.J., LAKSHMAN T.V., WONG L.: ‘SRED: stabilized RED’. Proc. INFOCOM, New York, US, May 1999, pp. 1346–1355

[12] TANG A., WANG J., LOW S.: ‘Understanding CHOKe: throughput and spatial characteristics’, *IEEE/ACM Trans. Netw.*, 2004, **12**, (4), pp. 694–707

[13] FLOYD S., GUMMADI R., SHENKER S.: ‘Adaptive red: an algorithm for increasing the robustness of red’s active queue management’. ACIRI, Technical Report, 2001

[14] https://www.cisco.com/en/US/docs/ios/11_1/feature/guide/WRED.html, accessed October 2009

[15] HU C., LIU B.: ‘Design of modified red implemented in a class based CIOQ switch’. Proc. 16th Int. Conf. on Computer Communication, Beijing, China, September 2004, pp. 655–660

[16] KORTEBI A., OUESLATI S., ROBERTS J.: 'Cross-protect: implicit service differentiation and admission control'. Proc. IEEE HPSR, Phoenix, US, 2004, pp. 56–60

[17] SALLES R.M., BARRIA J.A.: 'Proportional differentiated admission control', *IEEE Commun. Lett.*, 2004, **8**, (5), pp. 320–322

[18] SHREEDHAR M., VARGHESE G.: 'Efficient fair queueing using deficit round-robin', *IEEE/ACM Trans. Netw.*, 1996, **4**, (3), pp. 375–385

[19] GUO C.: 'Improved smoothed round robin schedulers for high-speed packet networks'. Proc. IEEE INFOCOM, Phoenix, US, May 2008, pp. 906–914

[20] PAREKH A.K., GALLAGER R.G.: 'A generalized processor sharing approach to flow control in integrated services networks: the single node case', *IEEE/ACM Trans. Netw.*, 1993, **1**, (3), pp. 344–357

[21] BENNETT J.C.R., ZHANG H.: '$\mathrm{WF}^2\mathrm{Q}$: worst-case weighted fair queuing'. Proc. IEEE INFOCOM, San Francisco, US, March 1996, pp. 120–128

[22] ZHANG L.: 'Virtual clock: a new traffic control algorithm for packet switching networks'. Proc. ACM SIGCOMM, Philadelphia, US, August 1990, pp. 19–29

[23] GOLESTANI S.J.: 'A self-clocked fair queueing scheme for broadband applications'. Proc. IEEE INFOCOM, Toronto, Canada, 1994, pp. 636–646

[24] JOUTSENSALO J., VIINIKAINEN A., HAMALAINEN T., WIKSTROM M.: 'Pricing based adaptive scheduling method for bandwidth allocation', *AEU-Int. J. Electron. Commun.*, 2007, **61**, (2), pp. 118–126

[25] LIU Y., KWOK Y.K., WANG J.: 'An adaptive packet scheduling algorithm for efficient downlink bandwidth allocation in UWB based wireless infrastructure networks', *Comput. Commun.*, 2007, **30**, (9), pp. 2087–2095

[26] SALLES R.M., BARRIA J.A.: 'Utility-based scheduling disciplines for adaptive applications over the internet', *IEEE Commun. Lett.*, 2002, **6**, (5), pp. 217–219

[27] Agere Inc.: 'Technical guide to the APP550TM and APP530TM traffic manager', white paper, 2004

[28] SUTER B., LAKSHMAN T., STILIADIS D., CHOUDHURY A.: 'Buffer management schemes for supporting tcp in gigabit routers with per-flow queuing', *IEEE J. Sel. Areas Commun.*, 1999, **17**, (6), pp. 1159–1169

[29] RAMAKRISHNA M.V., FU E., BAHCEKAPILI E.: 'Efficient hardware hashing functions for high performance computers', *IEEE Trans. Comput.*, 1997, **46**, (12), pp. 1378–1381

[30] http://dragonlab.org/traffic/, accessed July 2007

[31] http://pma.nlanr.net, accessed July 2007

[32] CHUANG S.T., GOEL A., MCKEOWN N., PRABHAKAR B.: 'Matching output queueing with a combined input output queued switch'. Proc. IEEE INFOCOM, New York, USA, March 1999, pp. 1030–1039

[33] CORMEN T.H., LEISERSON C.E., RIVEST R.L., STEIN C.: 'Introduction to algorithms' (The MIT Press, 2001, 2nd edn.)

10 Appendix

Proof of Lemma 1: SL_k is denoted as the average search length of a randomly built BST with k nodes, and it is obvious that $\mathrm{SL}_0 = 0$ and $\mathrm{SL}_1 = 1$. When $k \geq 2$, it can be formulated as

$$\frac{1}{k}\sum_{i=0}^{k-1}\frac{1}{k}[1 + i(\mathrm{SL}_i + 1) + (k - i - 1)(\mathrm{SL}_{k-i-1} + 1)] \quad (6)$$

Each term $0 \cdot \mathrm{SL}_0, 1 \cdot \mathrm{SL}_1, \ldots, (k-1) \cdot \mathrm{SL}_{k-1}$, appears twice in the summation, so we have the recurrence

$$\mathrm{SL}_k = 1 + \frac{2}{k^2}\sum_{i=1}^{k-1} i\,\mathrm{SL}_i, \quad k \geq 2 \quad (7)$$

Since, we further have

$$\sum_{i=1}^{k-1} i\,\mathrm{SL}_i = (k-1)\,\mathrm{SL}_{k-1} + \sum_{i=1}^{k-2} i\,\mathrm{SL}_i \quad (8)$$

Therefore

$$\mathrm{SL}_k = \left(1 - \frac{1}{k^2}\right)\mathrm{SL}_{k-1} + \frac{2}{k} - \frac{1}{k^2} \quad (9)$$

Given $\mathrm{SL}_1 = 1$, the following equation can be achieved

$$\mathrm{SL}_k = \frac{2}{k} - 1 + 2\left(1 + \frac{1}{k}\right)\sum_{i=2}^{k}\frac{1}{i} \quad (10)$$

In conclusion,

$$\mathrm{SL}_k = \begin{cases} 0, & k = 0 \\ 1, & k = 1 \\ \dfrac{2}{k} - 1 + 2\left(1 + \dfrac{1}{k}\right)\displaystyle\sum_{i=2}^{k}\frac{1}{i}, & k \geq 2 \end{cases}$$

□

Proof of Theorem 1: The expected value of successful search length is

$$E[\mathrm{SL}_k] = \sum_{k=0}^{l} \mathrm{SL}_k P_k \quad (11)$$

where SL_k is defined in Lemma 1. Substitute (2) and (1) into (11), and we can obtains the expected time of a successful search as follows

$$l\left(\frac{1}{M}\right)\left(1-\frac{1}{M}\right)^{l-1}+\sum_{k=2}^{l}P_k\left[\frac{2}{k}-1+2\left(1+\frac{1}{k}\right)\sum_{i=2}^{k}\frac{1}{i}\right]$$

□

Proof of Theorem 2: Still, P_k is defined in (2). The bound on the expected height H_k of a randomly built BST with k nodes has been verified in [33] as follows

$$H_k \le \log_2\binom{k+3}{3}-1 \tag{12}$$

Thus, the unsuccessful search length is

$$E[\mathrm{UL}]=\sum_{k=1}^{l}H_kP_k \tag{13}$$

which will be bounded by

$$\sum_{k=1}^{l}\binom{l}{k}\left(\frac{1}{M}\right)^{k}\left(1-\frac{1}{M}\right)^{l-k}\left[\log_2\binom{k+3}{3}-1\right]$$

□