Scalable Bandwidth Shaping Scheme via Adaptively Managed Parallel Heaps in Manycore-Based Network Processors

TAEHYUN KIM, JONGBUM LIM, JINKU KIM, and WOO-CHEOL CHO, Sogang University
EUI-YOUNG CHUNG, Yonsei University
HYUK-JUN LEE, Sogang University

Scalability of network processor-based routers heavily depends on limitations imposed by memory accesses and associated power consumption. Bandwidth shaping of a flow is a key function, which requires a token bucket per output queue and abuses memory bandwidth. As the number of output queues increases, managing token buckets becomes prohibitively expensive and limits scalability. In this work, we propose a scalable software-based token bucket management scheme that can reduce memory accesses and power consumption significantly. To satisfy real-time and low-cost constraints, we propose novel parallel heap data structures running on a manycore-based network processor. By using cache locking, the performance of heap processing is enhanced significantly and is more predictable. In addition, we quantitatively analyze the performance and memory footprint of the proposed software scheme using stochastic modeling and the Lyapunov central limit theorem. Finally, the proposed scheme provides an adaptive method to limit the size of heaps in the case of oversubscribed queues, which can successfully isolate the queues showing unideal behavior. The proposed scheme reduces memory accesses by up to three orders of magnitude for one million queues sharing a 100Gbps interface of the router while maintaining stability under stressful scenarios.

CCS Concepts: • **Networks → Routers**; • **Computing methodologies → Massively parallel algorithms**; • **Computer systems organization → Multicore architectures;**

Additional Key Words and Phrases: Manycore, network processor, token bucket, heap tree, adaptive control, stochastic modeling

ACM Reference Format:
Taehyun Kim, Jongbum Lim, Jinku Kim, Woo-Cheol Cho, Eui-Young Chung, and Hyuk-Jun Lee. 2017. Scalable bandwidth shaping scheme via adaptively managed parallel heaps in manycore-based network processors. ACM Trans. Des. Autom. Electron. Syst. 22, 4, Article 59 (May 2017), 26 pages.
DOI: http://dx.doi.org/10.1145/3065926

1. INTRODUCTION

To fulfill rapidly growing performance requirements, routers employ a network processor integrating tens or hundreds of processor cores. A manycore-based network processor performs packet processing, buffering, and scheduling. Its tight processing budget requires each core to complete its packet processing in thousands to tens of

This work was supported by ICT R&D program of MSIP/IITP. [B0101-16-0661, The research and development of the self-adaptive software framework for various IoT devices].
In addition, this work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIP; 2016 R1A2B4011799).
Authors' addresses: T. Kim, J. Lim, J. Kim, W.-C. Cho, and H.-J. Lee (corresponding author), Department of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea; emails: {taebang86, limjongbum}@gmail.com, {ecl0038, woocheolc, hyukjunl}@sogang.ac.kr; E.-Y. Chung, Department of Electrical and Electronic Engineering, Yonsei University, Seoul, Republic of Korea; email: eychung@yonsei.ac.kr.

DOI: http://dx.doi.org/10.1145/3065926

thousands of clock cycles while it accesses various internal and external resources [Cisco 2011, 2014]. A major challenge in network processor design is to build a *scalable* high-performance system that can process several hundred Gbps of packet streams. This requires the use of bleeding edge memory technology, which results in huge memory bandwidth and power consumption and often limits the scalability of the system.

In the near future, high-performance routers will buffer and schedule millions of network flows, for example, TCP or UDP flow. To buffer those flows and provide differentiated quality of service (QoS) for each flow, a router maintains millions of output queues. Recent edge routers support up to 512K queues and the number scales with increasing interface bandwidth [Cisco 2011, 2014] to reduce processing cost per flow.

Routers implement various QoS functions to provide differentiated services for network flows. QoS functions include latency guarantee, minimum bandwidth guarantee, rate limiting (or bandwidth shaping), and bandwidth sharing for different flows. Rate limiting is one of the major functions that limits the bandwidth of a flow at the input and output side of routers. To protect a router from a misbehaving network flow, it can perform access control, that is, rate limiting, based on the input bandwidth of an incoming flow. In addition, a router uses rate limiting to smooth the traffic scheduled out of itself so that downstream routers do not overflow. There exist both similarity and differences between rate limiting and various scheduling algorithms, such as deficit round robin (*DRR*) [Shreedhar and Varghese 1995], weighted fair queueing (*WFQ*) [Parekh and Gallager 1993], or worst-case fair weighted fair queueing (WF^2Q) [Bennett and Zhang 1996]. First, both *DRR* and *WFQ* do not provide bandwidth shaping (or rate limiting) but do guarantee minimum bandwidth or provide bandwidth sharing. On the other hand, WF^2Q can limit the bandwidth of a flow to a predetermined value. However, WF^2Q requires inserting the new schedule time of a queue into a sorted tree, which is costly and has a time complexity of $O(logN)$. For this reason, rate limiting is often implemented as a separate function using a token bucket (which has a time complexity of $O(1)$) in routers and per-queue-based rate limiting is supported by the router operating system. For instance, CISCO ASR 9000 supports 512K queues, each of which is shaped by a token bucket [Cisco 2011].

In slightly different contexts, several researchers propose rate-limiting schemes for NICs of data center servers. However, they either guarantee only minimum bandwidth or do not address the scalability issue. Elastic switch guarantees minimum bandwidth of network flows in the data center context using a programming hypervisor [Popa et al. 2013]. However, it does not provide rate limiting. SENIC proposes scalable NIC for end-host rate limiting [Radhakrishnan et al. 2014]. It uses WF^2Q to implement 10,000 rate limiters for one-level scheduling entities or to implement hierarchical bandwidth sharing in which virtual machines (VMs) are sharing interface bandwidth. Each VM's bandwidth is rate limited but flows within a VM are not rate limited. Again, WF^2Q requires per-packet sorting, leading to a scalability issue, and hierarchical bandwidth sharing does not guarantee per-flow rate limiting.

Traditionally, rate-limiting network flows in routers or switches are performed using token buckets [Giladi 2008; Franklin et al. 2003; Varghese 2010]. Recently, EyeQ proposed network performance isolation using parallel token buckets for data center applications [Jeyakumar et al. 2013]. It can implement 10,000 rate limiters using token buckets. However, the number of rate limiters is limited due to an expensive token refill mechanism. To maintain bandwidth shaping precisely for millions of queues, we need one token bucket for each queue. A token bucket is filled periodically to guarantee a predetermined bandwidth. A token fill procedure requires a read-modify-write operation on memory for a token update. Frequent token fills reduce the burstiness of traffic scheduled out of routers because the amount of added tokens is small for each fill. However, it increases memory bandwidth and power consumption. For instance, if we fill token buckets every 10μs (e.g., adding 1B per 10 microseconds shapes a queue to 100KB/s) and there are one million queues, we need to fill one hundred billion times per second. This becomes worse as we increase the number of queues supported in the network processor to reduce the per-flow processing cost. As the token bucket information is stored in off-chip DRAM parts due to its size, read-modify-writes caused by token bucket updates for millions of queues consume prohibitively large memory bandwidth and power.

Developing a scalable high-performance token bucket management scheme presents several challenges. First, to support the increasing number of queues, a token fill mechanism should be scalable in terms of memory bandwidth and power consumption. Second, the cost of token fill should be small enough not to introduce any major performance degradation in packet processing. Third, the scheme should be flexible to deal with a large dynamic range for the number of queues and bandwidth and be able to deal with harsh corner cases that can affect the reliability and availability of routers.

In this article, we propose a scalable software solution to the token bucket fill problem, which is based on event-driven token fills managed with parallelized heaps running on many processor cores of the network processor. Major contributions of our work are summarized as follows.

—We propose a scalable software method for the token fill problem, which does not periodically fill token buckets. Instead, it computes a reschedule time for a queue that runs out of tokens and reschedules the queue at the time when its token bucket becomes positive. This can reduce memory accesses significantly by up to three orders of magnitude. In addition, power reduction can be achieved by the same level of magnitude, which is crucial in the modern network processor design.
—To process tens of millions of reschedule events per second under real-time and low-cost constraints, we propose novel parallelized heaps mapped on manycores in the network processor. By allocating these data structures as arrays and using cache locking [Vera et al. 2003; Puaut and Decotigny 2002; Campoy et al. 2001], the performance of event processing is enhanced significantly and more predictable, which is crucial for real-time processing.
—We quantitatively analyze the performance and memory footprint of the proposed software scheme using stochastic modeling. More specifically, we use the Lyapunov central limit theorem to accurately estimate the performance and cost of a large number of reschedule events to predict the behavior of the system under various different operating conditions.
—The proposed scheme provides an adaptive management of reschedule threshold values to limit the number of reschedule events in the case of severely oversubscribed queues. This can successfully isolate the queues showing unideal behavior.

The proposed scheme reduces memory accesses by up to three orders of magnitude for one million queues sharing a 100Gbps interface in routers. In addition, the proposed scheme and modeling technique can apply to a general class of problems in which periodic resource accesses consuming power and memory bandwidth can be translated into independently parallelized sorted events.

2. BACKGROUND

2.1. Baseline Network Processor Architecture

Our proposed software scheme runs on a network processor. The baseline network processor architecture includes hundreds of processor cores and accelerator blocks that speed up specific network functions, for example, scheduling. To store packets, forwarding tables, and attributes of network flows, it is equipped with external memories, such as DRAM, SRAM, and TCAM.

There are several commercial manycore-based network processors available in the market [Tilera 2016; C. Networks 2013; Intel 2005]. Our proposed scheme may use off-the-shelf multicore- or manycore-based network processors—for example, Tilera, Cavium, and Intel—as baseline architectures. They have up to hundreds of processor cores for fast packet processing and some of their processing power can be allocated to perform token bucket processing. For instance, the Tilera NPS400 network processor has 256 cores and supports 24 channels of 16b DDR3/4 DRAMs [Tilera 2016]. The Cavium OCTEON III network processor has 48 cores and supports 4 channels of DDR3 DRAMs running at 1066MHz or DDR4 DRAMs running at 1200MHz [C. Networks 2013]. The CISCO flow processor employs 40 processor cores [Cisco 2014].

These network processors are equipped with dedicated components to speed up some functions of network processing. For instance, high-speed packet scheduling or high-speed destination lookup can be offloaded to dedicated components. The token bucket fill process can be implemented in a dedicated component because its basic operation is simple enough to be implemented as the read-modify-write of a token bucket value. In general, the disadvantages of dedicated components are that its design should be simple enough to be implemented in hardware and is not flexible with respect to changing input requirements or environments. For instance, the same network processor can be used for both core and edge routers where the number of supported queues are drastically different. If the execution time of a portion of packet processing depends on the number of queues, the dedicated hardware should be designed to cover the worst-case queue number, whereas the software solution can easily trade the supported queue number for the number of features processed in packet processing.

In our work, we use the baseline network processor architecture mentioned earlier and compare our software-based approach with the periodic token fill mechanism using a dedicated component.

2.2. Bandwidth Shaping via Token Bucket Management in Network Processor

Modern routers shape (or limit) the bandwidth of flows because many flows share an interface, for example, 100Gbps Ethernet. A flow is mapped on an output queue in routers and the bandwidth of a flow is shaped using a token bucket assigned to the associated queue. This packet flow handling is performed by a network processor of the routers. Figure 1 shows a packet flow inside a network processor. An incoming packet is stored in the packet buffer, which is indicated by (1) in the figure. A packet is assigned to a processor core, input processed, for example, destination lookup, and stored into an output queue. When the output queue containing the packet is scheduled, the packet is output processed by a core. These input and output processings are shown by (2). At the end of output processing, the token bucket of an associated queue is updated based on the scheduled packet size and the packet gets sent out of the router, marked in the figure by (3) and (4), respectively.

A token bucket is a simple mechanism used to provide bandwidth shaping of a packet flow [Varghese 2010]. Whenever a packet is scheduled, tokens equivalent to the size of a packet are deducted from the token bucket. To limit bandwidth, a predetermined amount of tokens are regularly added to a token bucket. Two key parameters of a token bucket mechanism, *committed information rate (CIR)* and *committed burst excess (CBE)*, are discussed in Franklin et al. [2003]. *CIR* is a long-term average bandwidth guaranteed to a traffic flow. If B bytes are added to the token bucket for a queue every T seconds, *CIR* becomes $\frac{B}{T}$ bytes/sec. *CBE* is a permitted short-term burst over the *CIR*. It refers to the maximum amount of tokens accumulated per flow (or queue), which limits the burst of traffic scheduled out of an associated queue.

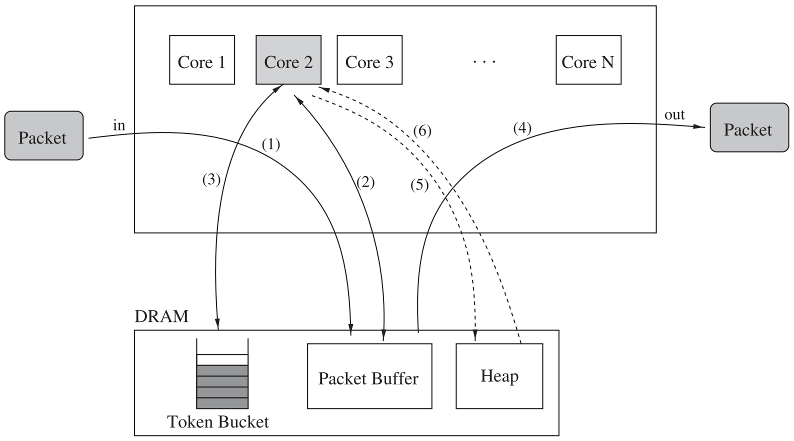


Fig. 1. Conventional system performs (1) storing a packet in a packet buffer, (2) assigning a packet to a core for input and output processing, (3) updating a token bucket after scheduling, and (4) sending out a packet. (5) and (6) are new additional processing by the proposed scheme, which includes insertion and deletion operation with heap trees.

In a conventional system, a token bucket update happens in three cases: initial token fills, periodic token fills, and token updates on scheduling packets. First, initial token fill occurs when the system is booted up to configure each queue with its predetermined queue bandwidth. Second, periodic fill occurs to fill tokens regularly to provide correct bandwidth. Finally, when a packet is scheduled out of a queue, tokens equivalent to the size of a scheduled packet are deducted.

Physically, as shown in Figure 1, token buckets and associated attributes are stored in external memories, for example, DRAM, because the amount of data does not fit in on-chip memories. In the network processor, parallel cores process (or schedule) packets and they access token buckets and attributes for update. Conventional token bucket management can be implemented in hardware or software.

The formula for the token bucket update upon a packet schedule and upon a token fill are shown in Equations (2) and (3), respectively.

$$\Delta TB_i = (T_{now} - T_{lu_i}) \times BW_i \tag{1}$$

$$TB_i = min(TB_i - Psize + \Delta TB_i, \overline{TB_i}) \tag{2}$$

$$TB_i = min(TB_i + \Delta TB_i, \overline{TB_i}) \tag{3}$$

In Equation (1), T_{now} and T_{lu_i} are current time and last token update time for queue i, respectively. BW_i represents the bandwidth of queue i, which is determined by CIR. Last update time is used to record the last time when the token bucket was updated [Franklin et al. 2003]. By subtracting the last update time (T_{lu_i}) from current time (T_{now}) and multiplying that with BW_i, additional tokens (ΔTB_i) that should have been accumulated in bytes since the last update time can be calculated and added to the token bucket.

In Equations (2) and (3), TB_i and $\overline{TB_i}$ represent the value of a current token bucket for queue i and its maximum value, respectively, and $Psize$ represents a packet size. $\overline{TB_i}$ is determined by CBE to limit the maximum burstiness. The difference between Equations (2) and (3) is $Psize$, as it represents a case in which a packet schedule causes a token update.

2.3. Related Work

Generic token bucket management schemes for network processors to guarantee QoS requirements are introduced in various texts [Giladi 2008; Franklin et al. 2003]. More advanced token bucket management methods are further discussed in Tang and Tai [1999], Park and Choi [2003], and Kidambi et al. [2000]. In Tang and Tai [1999], optimization of maximum burstiness and fill rates of token buckets are discussed to minimize the packet delay and queueing. In Park and Choi [2003], Kidambi et al. [2000], Aeron [2010], and AlQahtani [2015], adaptive/dynamic token bucket management schemes are presented. In Park and Choi [2003] and Kidambi et al. [2000], they attempt to dynamically change token bucket rates to provision fair shares under the case of oversubscribed or undersubscribed links. In Aeron [2010], fuzzy logic is used to vary token rate smoothly to improve bandwidth utilization. In AlQahtani [2015], they dynamically change the token rate to maximize the bandwidth utilization of heterogeneous wireless networks. In Chakravarthi and Shilpa [2013], multiple token buckets are used to improve bandwidth utilization in distributed networks. However, these prior token bucket management schemes do not address the scalability issue of token buckets in terms of memory bandwidth and power consumption but discuss how to assign token rates dynamically to maximize bandwidth utilization.

Manycore-based network processor design for routers has been an active research area. State-of-the-art designs are discussed in Cisco [2011, 2014], Tilera [2016], C. Networks [2013], and Intel [2005]. Intel IXP in Intel [2005] uses cores as pipelined computing elements to process a packet, whereas network processors in Cisco [2011, 2014], Tilera [2016], and C. Networks [2013] use tens to hundreds of cores as parallel computing elements, which provide more flexible programmability than Intel [2005]. Manycores in network processors perform packet processing, buffering, and scheduling, whereas some time-critical functions are offloaded to IP blocks or accelerators. While these works focus on the hardware structure of a network processor, others attempt to optimize the performance and cost of packet processing tasks mapped on manycores of the network processor [Huang and Wolf 2008; Ennals et al. 2005; Han et al. 2010]. In Huang and Wolf [2008], both parallel and pipelined design are modeled using queueing theory and their performances are compared under various traffic conditions. Ennals et al. [2005] propose a compiler technique that automatically partitions a network application into pipelined and parallel tasks mapped on a network processor. In Han et al. [2010], parallelized packet processing tasks are mapped on a GPU. Load balancing on multicore processors to optimize power consumption is discussed in Jeon et al. [2010]. Although these previous works discuss mapping network application tasks and load balancing them, none address the scalability issue of token bucket management.

3. PROPOSED SCHEMES

3.1. Scalable Event-Driven Token Fill Scheme with Parallelized Heaps

3.1.1. Event-Driven Token Fill. To avoid periodic fills and reduce power consumption and memory accesses, we propose a software method to perform event-driven token fills. The event-driven token fill refers to a token fill mechanism in which we update a token bucket only when there is an opportunity, such as a packet schedule.

In a conventional method, if a token bucket becomes negative due to a packet schedule, a queue becomes *unschedulable*. The queue becomes *schedulable* again when the next periodic fill happens and makes the token bucket positive.

One of the key ideas in the proposed method is to avoid periodic token fills and perform a token fill only when it is absolutely needed. As was pointed out earlier, token buckets are updated for three cases in the conventional method. Since we remove

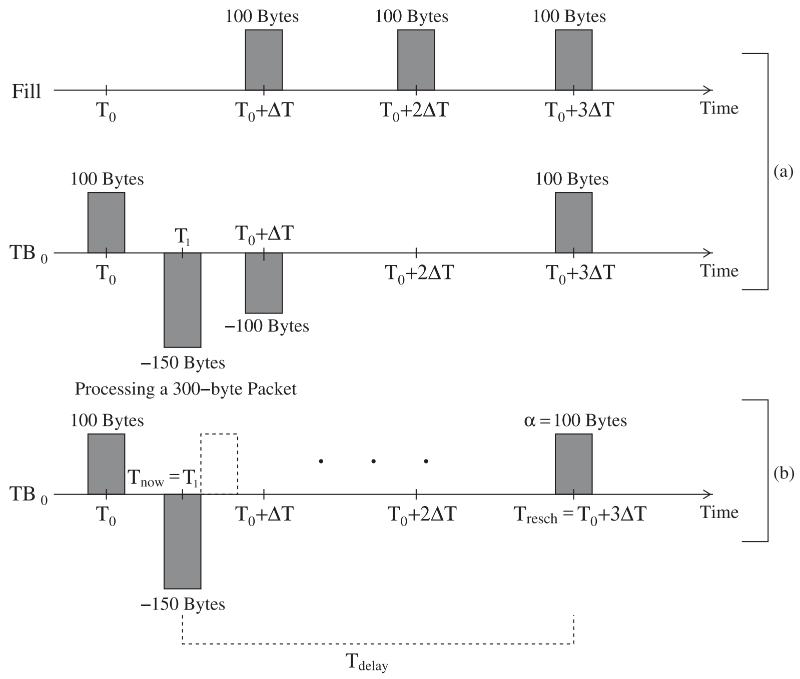


Fig. 2. Illustration of the difference between the (a) periodic fill scheme and (b) event-driven fill scheme.

periodic fills, the only case in which we update a token bucket after initialization is when we schedule a packet. However, because we do not fill the token bucket regularly, we should provide a method to make the token bucket positive when it becomes negative, that is, a queue is unschedulable. To achieve this, we compute a reschedule time that makes the token bucket of an unschedulable queue positive and put the associated output queue number and the reschedule time into a heap data structure that is sorted based on the reschedule time. As time progresses, the unschedulable queue (in the heap) that has the smallest reschedule time that is smaller than the current time is deleted from the heap, woken up, and becomes schedulable again. We call this method an event-driven token fill.

Figure 2 illustrates the difference between the periodic and event-driven fill schemes in terms of token bucket updates. Initially, at time T_0, a token bucket is initialized to 100B. A periodic fill scheme fills 100B at an interval of ΔT. At time $T_1 = \frac{1}{2}\Delta T + T_0$, a 300B packet is scheduled and reduces its token bucket to −150B because the token bucket value at T1 is 150B and the number of bytes scheduled at T1 is 300B.

Next, three fills update the token bucket and increase it back to 100B. On the other hand, an event-driven fill scheme disables the queue upon a token bucket becoming negative and computes the reschedule time that allows the token bucket to accumulate enough tokens to reach a reschedule threshold α. A token bucket is updated only once when it becomes negative. It is clearly shown that an event-driven fill scheme reduces memory accesses due to token bucket updates.

The event-driven token fill consists of four steps: initial token fill, token deduction upon a packet schedule, insertion into a heap, and deletion from a heap. Compared with the conventional token bucket scheme, a step for the periodic fill is replaced with a step for insertion into a heap and a step for deletion from a heap, which are described in Algorithm 1. It describes updating a token bucket and processing a heap after a packet

ALGORITHM 1: Token Bucket and Heap Management

```
1  Loop
       /* Packet processing and token update                                     */
2      if A scheduled packet is assigned to $Proc_j$
3      && ready for the transmission then
4          Get the queue number $i$ for a packet ;
5          Perform necessary packet processing ;
6          $TB_i = min( TB_i - Psize + (T_{now} - T_{lu_i}) \times \mathrm{BW}_i, \overline{TB_i})$; /* update token bucket */
7          if $TB_i$<0 then
8              $T_{delay} = \frac{\alpha_i - TB_i}{BW_i}$ ;
9              $T_{resch} = T_{now} + T_{delay}$ ;
10             $T_{lu_i} = T_{resch}$ ;
11             $TB_i = \alpha_i$ ;
12             Disable $queue_i$ ;
13             Insert($heap_j, i, T_{resch}$) ;
14         end
15     end
16     while reschedule time of root <current time do
           /* repeat the deletion until all roots with smaller schedule time
           exhausted.                                                            */
17         $i$ = Delete($heap_j$) ;
18         Enable $queue_i$ ;
19     end
20 EndLoop
```

is selected by the scheduler for transmission. After general output, packet processing is performed (lines 4 and 5) and a token bucket is updated in line 6. If a token bucket becomes negative (or unschedulable) as a result of a packet schedule, the reschedule time for the queue i is computed by Equations (4) and (5), which are also shown in lines 8 and 9.

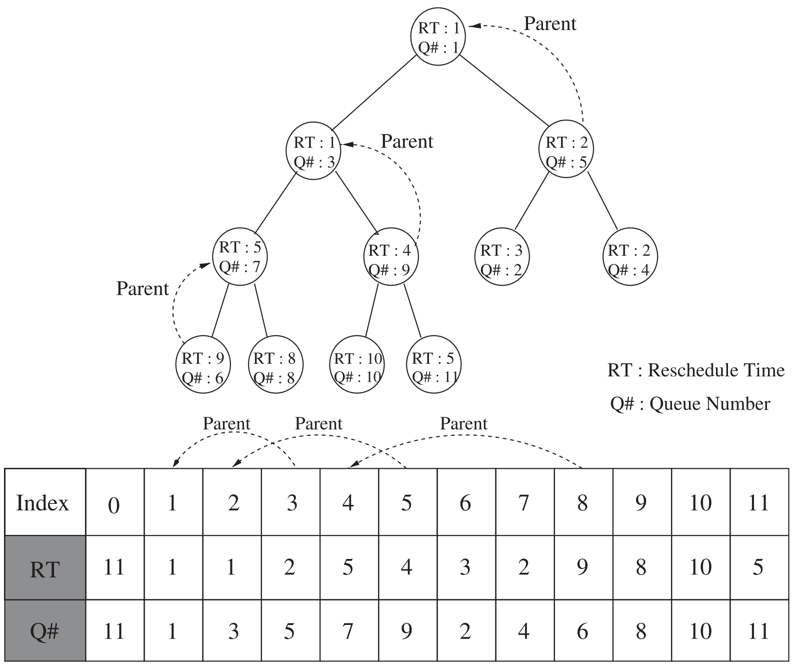
$$T_{delay} = \frac{\alpha_i - TB_i}{BW_i} \tag{4}$$

$$T_{resch} = T_{now} + T_{delay} \tag{5}$$

$$T_{lu} = T_{resch} \tag{6}$$

$$TB_i = \alpha_i \tag{7}$$

T_{delay} is the delay between current time and reschedule time. α_i and BW_i are the *reschedule threshold* value and *CIR* for queue i, respectively. When TB_i becomes negative, a queue becomes unschedulable. Then, at least tokens as much as $-TB_i$ should be accumulated before it becomes schedulable again. To avoid cases in which the value of a token bucket switches between negative and zero too often, we use a reschedule threshold, α_i, which is a positive number so that at least $\alpha_i - TB_i$ bytes are accumulated before the queue becomes schedulable. Because of this, TB_i does not become negative immediately after being schedulable again. $\frac{\alpha_i - TB_i}{BW_i}$ becomes T_{delay} and it is added to the current time, T_{now}, to determine the reschedule time, as shown in Equation (5). Finally, TB_i is set to α_i and the queue is disabled, as shown in lines 11 and 12. A computed reschedule time is used as a sorting key in the heap and inserted into a heap (for processor core j) with its queue number, as shown in line 13. At the end of each packet processing, all queues whose reschedule time is less than the current time are



Index	0	1	2	3	4	5	6	7	8	9	10	11
RT	11	1	1	2	5	4	3	2	9	8	10	5
Q#	11	1	3	5	7	9	2	4	6	8	10	11

Fig. 3. A heap data structure used in the proposed scheme.

made schedulable again, which is shown in lines 16 through 19. A clock that tracks the current time can be implemented with a simple cycle counter. This event-driven token fill scheme is much more scalable than periodic fill or WF^2Q because token buckets rarely become negative for bandwidth controlled flows, such as TCP flows; thus, the frequency of heap insertions or deletions is usually small.

3.1.2. Heap Data Structure for Sorting Unschedulable Queues. A heap is a binary tree data structure that contains elements with sorted keys, as shown in Figure 3. A heap is called a min or max heap according to elements sorted in ascending or descending order. In the proposed scheme, a reschedule time (RT) value is used as a sorting key. One useful characteristic of a heap is that a heap of size n that contains n elements can be allocated as an array of $n + 1$ elements, as shown in Figure 3. The first element in the array with index 0 is a special element and stores the current size of a heap. Array elements from indices 1 through 11 represent real heap elements. Because a heap can be built with an array, managing heaps has several benefits [Horowits et al. 1992]. First, although a heap is a binary tree technically, we do not need to maintain pointers. This saves space for nodes significantly. Second, a heap has a very simple traversal mechanism. Assuming that a node has an index of i, its left and right child can be accessed using indices of $2i$ and $2i + 1$, respectively, and its parent can be accessed using an index of $\lfloor \frac{i}{2} \rfloor$. This makes tree traversal and node management very simple. Third, elements are allocated sequentially as an array. Thus, if its maximum size is predetermined, it can be allocated as a chunk of contiguous memory and can be pinned in the portion of a cache memory using techniques such as cache locking.

In the proposed scheme, a min heap is used to store the reschedule times of unschedulable queues. When a queue becomes unschedulable due to the exhaustion of tokens, its reschedule time is computed and inserted into a min heap. For instance, assume that we have 11 elements in the heap, as shown in Figure 3. Upon adding a new queue, it is added to the array element position for index 12. The new queue's reschedule time is compared with its parent's value and the values are swapped if it is smaller than the parent's value. The swapping will be repeated until the newly added queue is placed in the right position of a heap. This process does not require inserting a new element in an array. The insertion time complexity is $O(\log(n))$, where n is the number of elements in a heap. The reschedule time of a root is compared with current time at the end of each packet processing, and a root is deleted from a heap when its reschedule time is smaller than the current time. Root removal has a time complexity of $O(log(n))$. It continues until no node with its reschedule time less than the current time is found. Again, this process does not require removing an element from an array. The size of n can affect the performance if it does not fit in the cache of a processor core.

3.1.3. Mapping Parallelized Heaps on Manycores and Optimizing Memory Hierarchy. In the proposed scheme, to speed up the insertion and deletion of a heap, we partition the heap and map it on parallel processor cores. Insertion into a heap is triggered by scheduling of a packet. Assuming that a packet is scheduled out of queue i, its packet processing is performed by a randomly assigned processor core j and is followed by the token bucket update. The queue number i is inserted into a heap managed by the core j if the token bucket becomes negative upon a packet schedule. Since the packet is randomly assigned to processor core j, each core has roughly the same number of unschedulable queues in its heap.

Partitioning a heap and mapping it on multiple cores offers several important benefits. Each partitioned heap mapped on a single core can process insertion and deletion of nodes independently with respect to other cores. As long as a node satisfies the heap property within its own heap, a node may be inserted into any heap. Therefore, many different partitions can exist and mapping can be flexible depending on the optimization goal, for example, load balancing. Second, heap partitioning does not require interprocessor communications among cores because insertion and deletion do not depend on operations on other heaps. This independence is a crucial property to relieve the complexity to maintain cache coherency and a locking mechanism. Third, dividing n nodes into roughly p groups of $\frac{n}{p}$ nodes can lead to execution time reduction because the reduced size of data can fit in the data cache for fast access. Finally, once the array implementing a heap is allocated on the cache, it does not have to be flushed out from the cache. When a new queue is inserted into a heap or a root is removed from the heap, only the values written into the array elements change and the current size of a heap is modified. Thus, cache locking shown in Figure 4 can be effectively used to allocate the space for a heap. If the size of a locked cache portion is L, $\frac{L}{node\ size}$ of nodes are permanently pinned on the cache and guarantee steady performance of insertion or deletion process. The nodes not covered by the size of L can be allocated dynamically on the off-chip memory.

3.2. Modeling the Number of Reschedules and the Size of Heaps

3.2.1. Modeling for a Single Queue. Estimating the number of reschedules and the size of a heap under various configurations and input traffic patterns are crucial in terms of accessing the performance and cost impact of the proposed solution under various stressful scenarios. To estimate the number of reschedules generated and the average size of a heap per processor core, we resort to stochastic modeling. Once the statistics for the packet size and interpacket arrival time of different flows are collected, we can use them to estimate the number of reschedules generated (performance overhead) and the average size of a heap per processor core (memory footprint).

First, we estimate how long a single queue is in the unschedulable state because the number of queues in the unschedulable state at any given time is the size of a heap. In

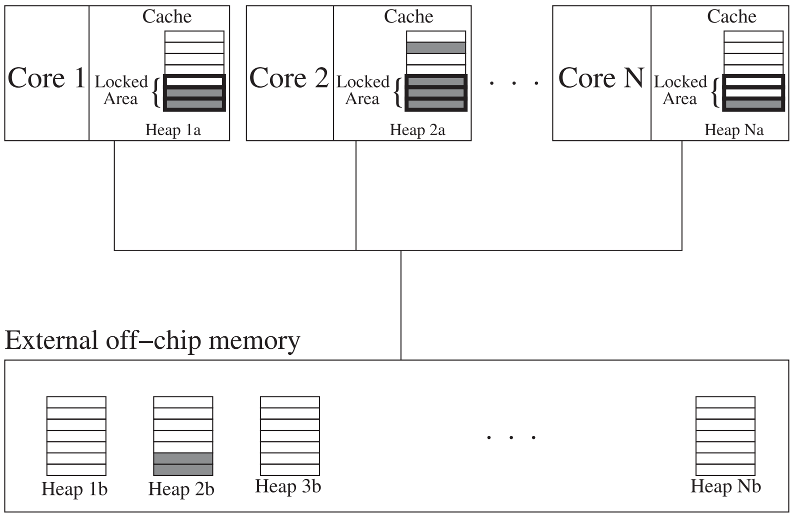


Fig. 4. Parallelized heaps partitioned onto locked cache portion and off-chip memory.

addition, the sum of schedulable and unschedulable periods can be used to compute the number of reschedules per second. To model the period stochastically, we need to know the packet size distribution and interpacket arrival time distribution for the queue.

If we denote M as a random variable for the number of reschedules per second, it can be expressed as

$$M = \frac{1}{N \times T_{int} + \frac{\alpha}{BW}}, \tag{8}$$

where N is a random variable for the number of packet arrivals making a token bucket negative and T_{int} is a constant interpacket arrival time. Calculating the mean and variance of M is discussed in the Appendix.

The ratio between schedulable and unschedulable periods of a queue determines the duration of a queue being in a heap. Let R denote a random variable for the ratio between the schedulable period and the sum of schedulable and unschedulable periods, as shown in Equation (9). Then, a random variable Y that denotes the chance of a queue being in the unschedulable state can be expressed in Equation (10).

$$R = \frac{N \times T_{int}}{N \times T_{int} + \frac{\alpha}{BW}} \tag{9}$$

$$Y = \begin{cases} 0, & 0 \le y < R \\ 1, & R < y \le 1 \end{cases} \tag{10}$$

The mean and variance of Y are given in Equations (11) and (12), respectively.

$$\begin{aligned} E(Y) &= 0 \times E(R) + 1 \times E(1-R) \\ &= 1 - E(R) \\ &= 1 - \sum_{n=1}^{n_{max}} R \times P(N=n) \end{aligned} \tag{11}$$

$$\begin{aligned} Var(Y) &= E(Y^2) - (E(Y))^2 \\ &= (1 - E(R)) - (1 - E(R))^2 \\ &= E(R)(1 - E(R)) \end{aligned} \tag{12}$$

The mean and variance of a random variable Y for a single queue are used to estimate the behavior of many queues in the next section.

3.2.2. Modeling for Many Queues. To illustrate the estimation of the average heap size per core, we use an example. For 100 processors with 1,000,000 queues, each processor core is responsible for roughly 10,000 queues. Without losing generality, core j can process packets for queue 0 to 9,999. If we can assume that each queue sees the same traffic distribution (i.e., independent identical distribution for packet size and interpacket delay), we can write the probability density function using the central limit theorem because the number of queues is huge, for example, 10,000. In general cases, however, we see nonidentical distributions for the packet size and interpacket delay. If each queue sees a different distribution for the packet size and interpacket delay, we can use the Lyapunov central limit theorem to get the probability [Billingsley 1995]. The probability density function for the average heap size is shown in Equation (13), where X is a random variable for the heap size, σ is $\sqrt{\sum_{i=1}^{m} Var(Y_i)}$, μ_i is $E(Y_i)$, and m is the total number of queues per core. The equation indicates that the distribution is a rapidly decaying exponential function, with its mean equal to the average number of queues being in the unschedulable state.

$$P(X) = \frac{1}{\sqrt{2\pi}\sigma} \exp^{\frac{-(X - \sum_{i=1}^{m} \mu_i)^2}{2\sigma^2}} \tag{13}$$

Since a small number of flows represent most flows in terms of packet size and interpacket distribution, the average heap size can be readily estimated using Equation (13). Similarly, the estimation for the number of reschedules for many queues can be done by calculating the mean and variance of M in Equation (8) for representative queues and using the Lyapunov central limit theorem.

Pathological cases may arise in the proposed scheme and the size of the heap may get bigger than one that can fit in the locked area of the data cache. This can happen for unbalanced loads on processor cores or oversubscription of queues. We evaluate how much these abnormal operating conditions affect the performance by estimating the number of reschedules and size of heaps for severely overloaded queues in Section 5.2.1. Our proposed scheme relies on an adaptive threshold control method to deal with this problem, which will be discussed in the next section.

3.3. Adaptive Reschedule Threshold (α) Control to Isolate Queues Showing Unideal Behaviors

In the network processor, processor cores often work under a tight budget to process packets. Thus, additional processing due to heap management is translated into less computing resources available for packet processing. Under a normal operating mode, heap management accounts for a small fraction of total computation. However, if too many reschedules are generated for a short period or load balancing for heap processing among cores is broken, heap processing may affect packet processing performance. The proposed scheme adopts an adaptive feedback control loop [Åström and Wittenmark 2013; Zhang et al. 2002; Kang et al. 2012], which monitors the size of the heaps and dynamically updates the reschedule threshold, α_i, using adaptive feedback control shown in Figure 5. If too many reschedules are generated, we increase α_i. A larger α_i increases the number of packets to be scheduled before the token bucket

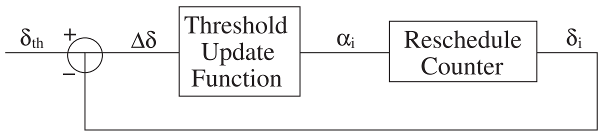


Fig. 5. Adaptive feedback control of reschedule threshold.

becomes negative again. Thus, a longer time elapses between two consecutive reschedule generations at the cost of burstiness, which reduces the total number of reschedules.

The proposed adaptive reschedule threshold control scheme uses Equations (14a) and (14b) to dynamically update the threshold:

$$\alpha_i = \begin{cases} \gamma \times \alpha_i, & \delta_i > \delta_{th} \qquad (14a) \\ max\left(\dfrac{\alpha_i}{\gamma}, \alpha_{i_{min}}\right), & \delta_i <= \delta_{th}, \qquad (14b) \end{cases}$$

where γ is a multiplicative factor that is multiplied to the reschedule threshold α_i, δ_i is a current reschedule count for the queue i during a current sampling period, δ_{th} is a threshold value to compare with the current reschedule count, and $\alpha_{i_{min}}$ is a minimum value of α_i. If δ_i is larger than δ_{th}, which indicates that too many reschedules are generated, a multiplicative factor is multiplied to increase the reschedule threshold. This will decrease the frequency of reschedules so that cores are not spending too much computing resources on heap processing. If δ_i is less than δ_{th}, we reduce the reschedule threshold, α_i. We experimentally optimize the parameters for the adaptive control scheme, such as sampling interval, γ, and δ_{th}.

4. EVALUATION METHODOLOGY

4.1. Simulation Environment

In general, general-purpose multicore (GPMC) servers are not adequate to generate input packet traffic, schedule packets, process token buckets, and evaluate metrics. First, in routers, packets (input traffic) are generated externally and arrived at routers while conforming to various interpacket delay and packet size distribution models. The input packets are processed by a memory controller and a scheduler as soon as they enter the router. However, in GPMC servers, packets are usually accumulated and processed in burst for efficiency by the OS. Thus, the token bucket management code running on these servers cannot see the interpacket delays that a router can see. Second, we use various metrics, such as the number of DRAM accesses, power consumption of the memory system, and heap processing time with various cache configurations. These are not easily measured by GPMC servers.

For this reason, an *in-house event-driven simulator* is implemented to simulate various configurations and test vectors at a higher speed, which can complement slow and accurate simulation with multicore or manycore processor simulators (e.g., GEM5). The simulator consists of an input traffic generator, processor cores performing token bucket updates and heap processing, and loggers, as shown in Figure 6. Knobs to control various input types and architectural parameters are provided. Various statistics—such as the number of reschedules, size of heaps, schedulable and unschedulable periods—are collected. The simulator also generates a memory trace file that logs memory operations due to heap insertion and deletion operations. These memory trace files are used for the DRAMSim2 simulator [Rosenfeld et al. 2011] to accurately measure the static and dynamic power consumption.

In addition to the event-driven simulator, we implement a *cycle-accurate simulator using GEM5* [Binkert et al. 2011; Martin et al. 2005] and verify the functionality

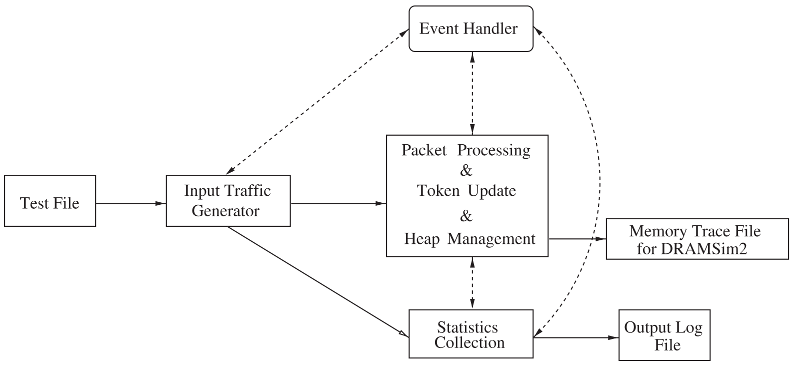


Fig. 6. An event-driven simulator.

and performance in a cycle-accurate environment. The simulator is configured to have processor cores running at 1GHz and each core has a 2-way 32KB first-level instruction and data cache. In addition, we implement cache locking to lock down 2KB, 4KB, and 8KB of the first-level data cache for heap data and measure the execution time in cycles and the number of cache misses for both insertion and deletion operations. To verify our proposed method, we implement the code described in Algorithm 1. We sweep the number of queues covered by one core from 1 queue to 10,000 queues. The size of each heap element is 8B to store a reschedule time and a queue number. We insert GEM5 instructions (e.g., m5_reset_stats and m5_dump_stats) into the code to measure statistics for inserting (deleting) an element into (from) the heap.

4.2. Metrics

Key metrics are defined and used to access the impact of performance and cost and confirm viability. First, *the number of DRAM accesses* is used to compare a periodic and event-driven fill scheme in terms of external memory accesses. Second, to determine the performance impact of the proposed scheme on packet processing, we use *the number of reschedules per second per core* and *execution time for an insertion or a deletion operation*. Each reschedule generates one insertion and one deletion operation. Thus, multiplying these two factors shows the performance impact. Third, *the size of a heap per core* is used to determine the footprint of memory usage by the heap. The size of a heap also affects performance, as it increases tree traversal time and cache misses.

4.3. Input Setting

The simulator has various knobs to control inputs and architectural parameters, including packet size distribution, average bandwidth, including overload factor, interpacket delay distribution, reschedule threshold, number of queues, and number of processor cores. In various experiments, we vary the input load. Input load of 1 indicates that the bandwidth of an input flow matches the output bandwidth allocated to its queue. However, depending on the packet size and interpacket delay distribution, overload can fluctuate temporarily and becomes larger than 1. The overload factor of a queue is defined as follows.

$$Overload = \frac{Input\ rate}{Output\ rate} \tag{15}$$

$$= \frac{Average\ packet\ size}{Average\ interpacket\ delay \times queue\ scheduling\ bandwidth} \tag{16}$$

Table I. IMIX Packet Size Distribution

Packet Size	Distribution Ratio (%)
28B	1.20
40B	35.50
44B	2.00
48B	2.00
52B	3.50
552B	0.80
576B	11.50
628B	1.00
770B	29.50
1420B	3.00
1500B	10.00

For instance, assume that the scheduling bandwidth of a queue is 1Mbps. If the average packet size is 500B and average interpacket delay is 1ms, its overload factor is $\frac{500bytes * 8bits/byte}{1\ msec \times 1\ Mbps} = 4$.

For the packet size, we use internet MIX (IMIX) [Agilent 2007] and uniform distribution from 40B to 1500B. Packet size distribution for IMIX is shown in Table I.

For the interpacket delay, we use constant delay and exponential distribution. For exponential distribution, λ is chosen so that $\frac{1}{\lambda}$, the mean of the exponential distribution (or the mean of interpacket delays), varies to cause overloading of an output queue from 0.9 to roughly 8 times, whereas the shaping bandwidth of an output queue varies for different configurations.

In addition, real traces are obtained from CAIDA [2008] and the data are extrapolated and used to create realistic scenarios.

5. RESULTS

5.1. Scalability Analysis

We measure *the number of DRAM accesses*, *the size of a heap per core*, and *Heap Processing Time (insertion and deletion time)* to evaluate the scalability of the proposed method.

5.1.1. Number of DRAM Accesses. The proposed method is compared with a periodic fill scheme in terms of external DRAM accesses. We compare only DRAM accesses due to fill operations. Each fill operation in the periodic fill scheme requires two memory accesses (read-modify-write). In the proposed scheme, there is no fill operation. Instead, we count the number of DRAM accesses due to insertion and deletion operations.

We configure the interface of a router as 100Gbps and vary the number of queues that share 100Gbps interface. In a system with 1,000 identical queues, the bandwidth of each queue is shaped to 12.5MB/s ($\frac{100\text{Gbps}}{1000} = 100\text{Mbps}$). For 1,000,000 identical queues, the bandwidth of each queue is shaped to 12.5KB/s ($\frac{100\text{Gbps}}{1000000} = 100\text{Kbps}$). Figure 7 shows the number of read-modify-writes due to periodic fills for different fill periods ranging from 100μs to 10ms with respect to different number of queues. For 1,000,000 queues, the number of memory accesses ranges from 2×10^9 to 2×10^{11} for 10s.

Table II shows the number of memory accesses due to insertion and deletion operations for 10s with respect to different queue numbers for the proposed scheme. The size of packets is generated based on the Internet Mix, where the average packet size is set to 515B and interpacket delays are exponentially distributed. The results show that the number of memory accesses with the locked portion of 8KB cache are from 414900 (1,000 queues) to 45274999 (1,000,000 queues). When we compare the result

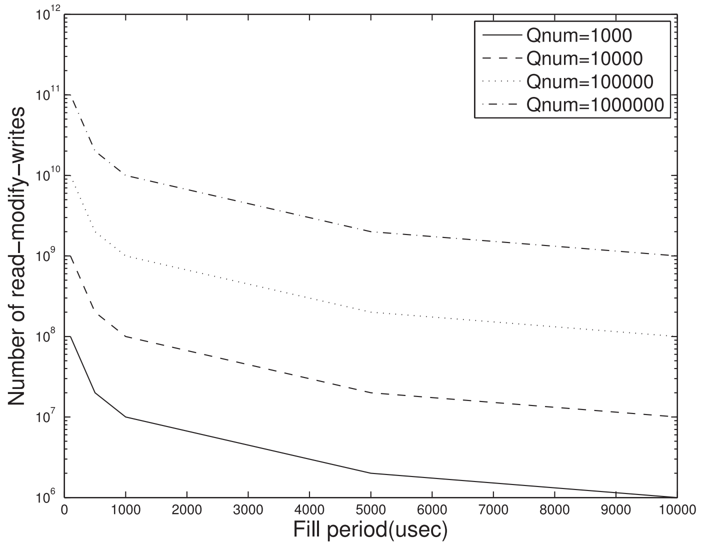


Fig. 7. Number of read-modify-writes for 10s with respect to different fill periods (from 100μs to 10msec) and number of queues (from 1,000 to 1,000,000) when the periodic fill scheme is used.

Table II. Number of DRAM Accesses Due to Heap Processing for 10 Seconds with Respect to Queue Numbers and Locked Cache Size (2KB, 4KB, 8KB)

Number of Queues		1000	10000	100000	1000000
Number of Accesses	2KB	503500	3896500	14232600	60928300
	4KB	483900	3475500	14233500	53088299
	8KB	414900	3255100	14274100	45274999
Number of Reschedules		34063	199075	748836	2273204

of the proposed scheme with the periodic fill scheme, the reduction gain ranges from $\frac{2\times10^{9}}{45274999} = 44.17$ to $\frac{2\times10^{11}}{45274999} = 4417.44$ for 1,000,000 queues, which shows that the proposed scheme is much more scalable.

The worst-case memory accesses, 6 million operations per second for 1,000,000 queues and 2KB cache, are possible with one channel DDR 3/4. If 6 million operations per second land on a single bank, the average latency ($\frac{1\ second}{6\ million\ accesses} \sim 166\ nsec$) between memory accesses is larger than typical T_{rc} of a DRAM bank. Compared with the proposed method, the conventional periodic fill scheme does not scale in the case of 1ms and 100μs refill periods because the bandwidth requirements are not feasible with current DRAM technologies.

5.1.2. Size of Heaps with Respect to Queue Numbers. This experiment evaluates the size of a heap per core for different queue numbers from 1,000 to 1,000,000 under internet packet size mix (IMIX) and exponential distributed interpacket delays.

Figure 8 shows the average and maximum heap size from cores 1 through 100 for different queue numbers assuming that 100 processor cores are used. A heap grows and shrinks as unschedulable queues are inserted and deleted. As parallelized heaps are spread over 100 processor cores, the size of a heap is roughly 630 nodes on average or 900 nodes at most per core for 1,000,000 queues. Assuming that each node occupies 8B to store the queue number and reschedule time, these numbers are translated into $8 \times 630B$, which can fit easily in the data cache. This implies that the heap data structure may be stored in external SRAM existing in the network processor, although

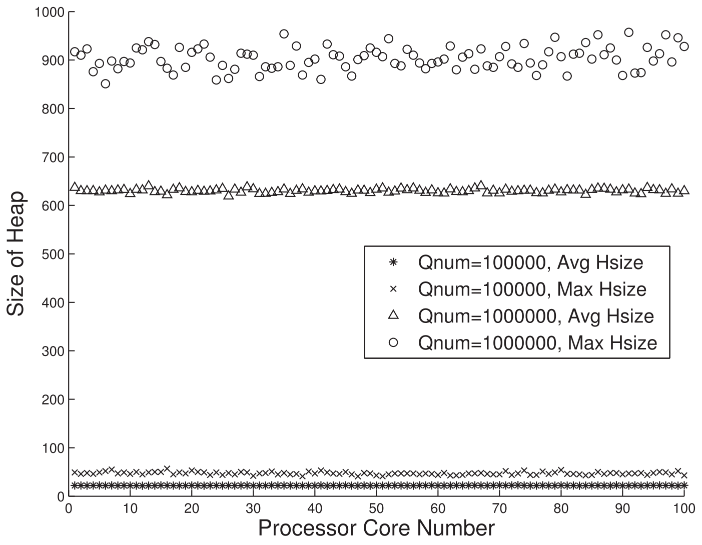


Fig. 8. Average and maximum size of a heap per processor core with respect to different number of queues (packet size = IMIX, interpacket delay = exponential, overload = 1, unit in y axis = number of nodes in a heap).

Table III. Specification of Memory System in Gem5

Types	No cache locking	Cache locking
L1 inst. cache	1MB, 512-way	32KB, 2-way
L1 data cache	1MB, 512-way	32KB, 2-way
Main memory	DDR3 1600MHz,1-channel, 2-rank/channel, 8-bank/rank	

we have to accurately verify the worst-case heap size. In some pathological case (e.g., link failure), all 1,000,000 queues run out of tokens and the worst-case heap size can be 1,000,000 nodes * 8B = 8MB. This still can be accommodated in external SRAM or DRAM. The decision to choose between SRAM or DRAM depends on price justification of using SRAM over DRAM and usable bandwidth and space available in the SRAM or DRAM module of the memory system.

5.1.3. Heap Processing Time Assuming No Cache Misses. The exact insertion and deletion times of a queue are measured using a modified GEM5 cycle accurate multi/manycore simulator. To figure out the performance of codes without any memory effect, we fetch all heap elements during initialization so that no cache misses occur during insertion and deletion. The specification of the baseline memory system is shown in Table III.

Tables IV and V show the insertion and deletion times, respectively, for different queue numbers in cycles under the assumption of an ideal memory system (no cache misses). Considering packet-processing time ranges from a few thousand to tens of thousands of cycles, the insertion or deletion time is relatively small. The insertion time ranges from 27 cycles to 33 cycles, which does not vary much with the heap size. When an insertion operation adds a new node to the leaf of a heap, it is likely to have a reschedule time bigger than those in the heap. Thus, swap operations rarely occur and insertion time remains relatively constant with respect to the number of queues. On the other hand, a delete operation removes the root node and inserts the largest node at the leaf into the root. This node traverses a heap all the way to the leaf and causes

Table IV. Execution Time for Insertion Operations (In cycles)

Number of queues	Overload factor				
	5.15	2.58	1.72	1.29	1.03
100	32.00	32.00	32.00	32.00	32.01
1,000	32.01	32.01	32.01	32.04	32.14
10,000	32.02	32.02	32.02	32.02	32.03
100,000	32.02	32.02	32.02	32.02	32.02
1,000,000	32.28	32.34	32.33	32.21	32.03

Table V. Execution Time for Deletion Operations (in cycles)

Number of queues	Overload factor				
	5.15	2.58	1.72	1.29	1.03
100	48.00	48.00	48.00	48.00	48.00
1,000	78.55	74.37	73.90	68.07	49.60
10,000	118.12	114.63	107.92	102.91	74.65
100,000	153.26	151.27	147.20	140.44	110.99
1,000,000	214.32	208.45	198.04	179.00	156.03

Table VI. Execution Time of Insertion and Deletion Operations with Cache Locking in Cycles (Packet size = IMIX, Interpacket Delay = Exponential, Overload Factor = 1.29)

Locking Size	Insert	Delete
2KB	446.7803	879.8279
4KB	448.5005	787.4605
8KB	449.1315	655.7430

the execution time proportional to the height of a heap. The result shows increased execution time with increasing number of queues and overload factor because the heap size increases.

5.1.4. Heap Processing Time With Cache Locking. The performance of insertion and deletion operations with cache locking are measured. Overload factor is set to 1.29 and 1,000,000 queues are used. IMIX and exponential distribution are used for the packet size and interpacket delay distribution, respectively. Table III shows the specification of the memory system under cache locking.

Table VI shows the performance of insertion and deletion operations. Unlike the performance under no cache misses, insertion operations cause relatively large latency, ranging from 446 to 449 cycles because heap nodes close to the leaf need to be fetched and half of memory accesses cause cache misses (shown in Table VII). The number does not vary much with increasing cache locking size since doubling or quadrupling size does not improve much in terms of the depth of the cached heap. On the other hand, deletion operationsimprove as the locked cache portion increases since deletion requires traversal of the entire heap. The execution time for insertion and deletion operations are tolerable, as the network processor adopts multithreaded cores that can hide cache misses.

5.1.5. Power Consumption Analysis. Tables VIII and IX show total and dynamic power consumption for the periodic fill scheme and our proposed scheme, respectively. They are measured using DRAMSim2 simulations. Total power consumption includes background (static), refresh, and dynamic power consumption. Dynamic power consumption includes power consumption only for processing read-and-write operations.

Table VII. Number of Cache Misses and Miss Rates with Different Cache Locking Sizes (Packet Size = IMIX, InterPacket Delay = Exponential, Overload Factor = 1.29)

Locking size	Insert		Delete	
	Average miss count	Average miss rate	Average miss count	Average miss rate
2KB	10.17	55.51%	13.91	12.23%
4KB	10.18	55.52%	11.94	10.47%
8KB	10.18	55.48%	9.94	8.72%

Table VIII. Total Power Consumption (in Watts) for the Periodic Fills and the Proposed Scheme)

Number of Queues	Periodic Fill (10ms period)	2KB	4KB	8KB
10,000	9.34	1.15	1.15	1.15
100,000	10.40	1.17	1.17	1.17
1,000,000	20.84	1.29	1.27	1.21

Table IX. Dynamic Power Consumption (in Watts) for the Periodic Fills and the Proposed Scheme)

Number of Queues	Periodic Fill (10ms period)	2KB	4KB	8KB
10,000	0.075	0.003	0.003	0.003
100,000	0.761	0.012	0.012	0.012
1,000,000	7.577	0.110	0.087	0.034

For the periodic fill scheme, to accommodate the bandwidth requirement, eight channels of DDR3 DRAMs are used and each channel has a 64b I/O interface running at 800MHz. We only include the results from cases in which the refill period is 10ms because some cases with shorter refill periods (1ms and 100μs) are not feasible. Our proposed scheme uses only one channel because it provides sufficient bandwidth.

The results for total power consumption show 8.12 (periodic fill vs. 8KB for 10,000 queues) to 17.22 (periodic fill vs. 8KB for 1,000,000 queues) times power reduction. For the small number of queues, the static power consumption dominates and the improvement seems smaller. Considering the fact that shorter refill periods would consume even larger power if more channels were to be allocated, our scheme is highly scalable considering power consumption. When we consider only dynamic power consumption, the results show up to 222.8 (periodic fill vs. 8KB for 1,000,000 queues) times reduction.

5.2. Evaluating under Various Scenarios

5.2.1. Overloaded Queues. Queues can be oversubscribed. This means that traffic can come in at a rate higher than the allocated bandwidth assigned to the associated queue. Figure 9 shows the total number of reschedules generated for all cores and the size of a heap per processor core for different overload factor values assuming that all queues are oversubscribed simultaneously (a pathological case). Four lines represent four different combinations of packet size and interpacket delay distribution.

To validate our stochastic model, we measure the heap size distribution using a simulator and compare the distribution obtained from our stochastic model. The results are shown in Figure 10. For the uniform distribution from 40B to 1500B, we use the overload factor of 1.1, 1.28, 1.54, 1.93, 2.56, 3.85, and 7.7. The overload value simply represents a case in which a queue is receiving packet traffic whose bandwidth is larger than its programmed token bucket bandwidth by the specified overload value. These overload factors are derived values from average packet sizes, interpacket delay,

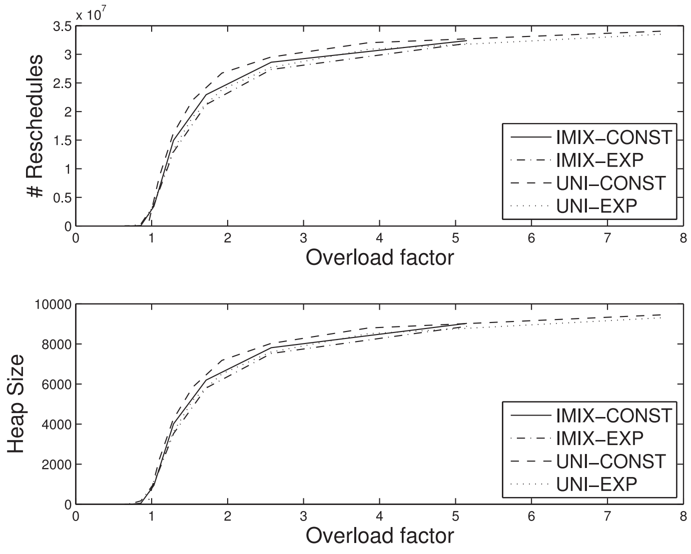


Fig. 9. Total number of reschedules generated and average size of a heap per core with respect to different overload factors.

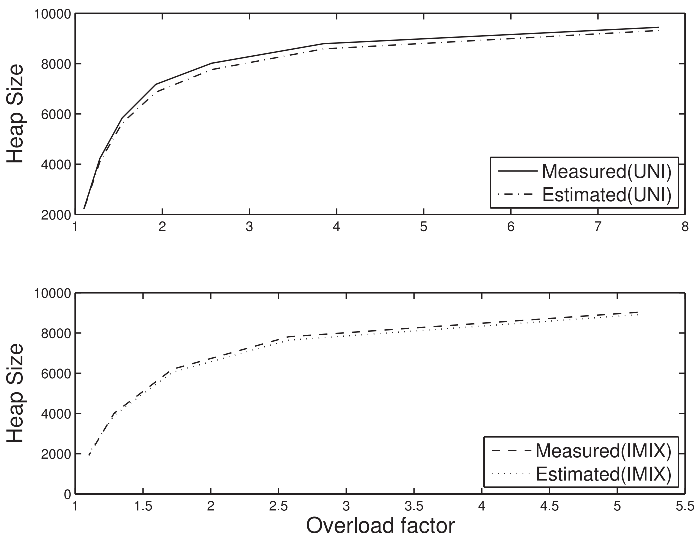


Fig. 10. Estimated and simulated heap size distribution per core for 1,000,000 queues (packet size = uniform (upper), IMIX (lower), interpacket distribution = constant).

interface bandwidth, and total number of queues. The upper plot shows its measured and estimated average heap size. The bottom plot uses an IMIX packet size distribution.

The difference in the average number of heap sizes between our model and simulated results for different overload factor values are summarized in Table X. It shows that the estimation of the model is quite accurate, as the mean of errors is only 2.73%.

5.2.2. Periodically Overloaded Queues. Figure 11 shows how the proposed adaptive control scheme works when input traffic is overloaded periodically. In this experiment, the input is not overloaded for first 10s. For the next 10s, the input is overloaded (overload

Table X. Errors for Heap Size Between the Proposed Model and Simulation Results

Overload (Uni)	7.7	3.85	2.56	1.93	1.54	1.28	1.1
Errors (Uni) (%)	1.3	2.5	3.3	4.6	3.6	3.0	1.0
Overload (Imix)	5.15	2.57	1.72	1.29	1.1	–	–
Errors (Imix) (%)	1.3	2.2	2.5	2.7	4.8	–	–

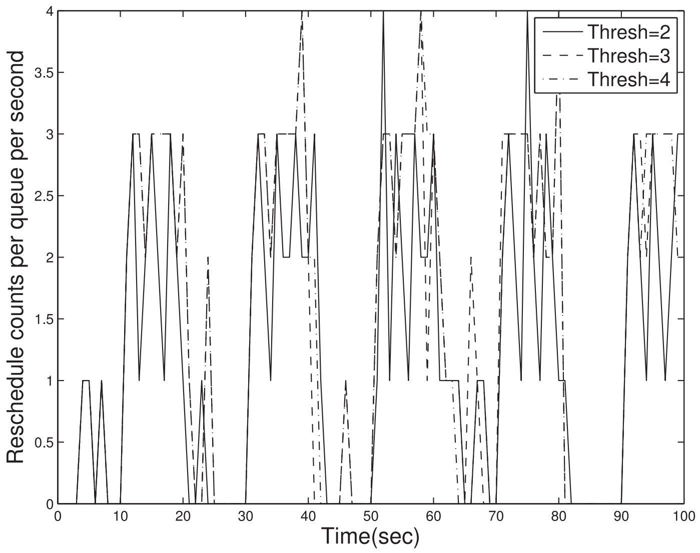


Fig. 11. Reschedule count variation under an adaptive reschedule threshold scheme with respect to different reschedule count threshold, δ_{th} (packet size = IMIX, interpacket delay = exponential, overload = 1 and 3.85).

factor = 3.85). This pattern is repeated during the whole simulation time. The result shows that a reschedule count increases during overloaded periods. Three lines represent three different values for the reschedule count threshold, δ_{th}, in Equations (14a) and (14b). When δ_{th} is 2, the reactive feedback loop responds aggressively and increases reschedule threshold α_i quickly, which results in less overall reschedule counts compared with two other threshold values 3 and 4. In all cases, the feedback loop forces the reschedule count to be within δ_{th} reasonably well.

5.2.3. Experiments with Real Internet Packet Data. To validate our scheme under realistic scenarios, we run simulations using real Internet packet data. The packet data are obtained from CAIDA [CAIDA 2008] (The Cooperative Association for Internet Data Analysis). Statistics for packet data are summarized in Table XI. These data are collected from two locations (Chicago, San Jose) and each location has two directions (Dir A, Dir B).

A noticeable difference between IMIX and real data is the average packet size. The average packet sizes for real data are bigger than IMIX. Table XI has the key statistics summary for real data, including the number of traffic flows, transmission rate, and average packet size. Average bandwidths for four different data range from 2.01Mbps to 2.28Mbps. Since packet data are for relatively small bandwidth interfaces, such as OC-48 and OC-192, we extrapolate the data and build realistic stressful scenarios. For instance, a 100Gbps interface is divided by average queue bandwidth determined from real data to compute the number of queues for the simulations. Because data do not contain interpacket delay information, we use both constant and exponential

Table XI. Summarized Statistics for Packet Flows in Real Packet Data Collected from CAIDA [2008]

Data set	Transmission rate	Flows	Average bandwidth	Average Packet Size
	bits/s	flows/s	bits/flow	bytes
Chicago (Dir A)	1370000000	6380	214733.5	642
Chicago (Dir B)	3410000000	16060	212328.8	882
San Jose (Dir A)	3690000000	16180	228059.3	852
San Jose (Dir B)	2570000000	12730	201885.3	759

Table XII. Average Heap Size and Number of Reschedules in Chicago Direction A

Overload factor		Constant Distribution		Exponential Distribution	
TCP	UDP	Average heap size	Number of Reschedules	Average heap size	Number of Reschedules
0.9	1.03	432.3	3457993	411.7	3127650
0.9	1.29	543.7	4368243	485.5	3718001
0.9	1.72	658.9	5298872	566.4	4356358
1.0	1.03	1162.0	9325105	919.5	7115239
1.0	1.29	1265.8	10148546	1020.1	7892747
1.0	1.72	1378.1	11025129	1115.0	5837917
1.1	1.03	1664.6	13278297	1396.2	10789900
1.1	1.29	1789.9	14242481	1493.9	11537981
1.1	1.72	1886.4	14978966	1583.2	12218941

distribution. Although we have performed experiments with data from two locations and various TCP and UDP traffic ratios, we show the result from only one experiment due to the size limit of this article.

In the experiment, we divide flows into TCP and UDP flows whose ratio mimics the ratio in real data. TCP approximately constitutes over 90% of packets in both Chicago and San Jose. We vary TCP average overload factor from 0.9 to 1.1 while UDP overload factors are set to slightly higher values (from 1.03 to 1.72), indicating some congestion. Table XII shows average heap sizes and the number of reschedules with respect to different TCP and UDP overload factors. The difference from our synthetic IMIX distributions is that both average heap size and reschedule counts are increased. To identify the cause of the difference, we compare an average queue occupancy during an unschedulable period, an average negative value for a token bucket (upon a queue being unschedulable), and an average packet size, which are shown in Table XIII. When the average overload is 1, the input bandwidth of a flow matches the scheduling bandwidth; thus, a queue may not have to go into an unschedulable state. However, a queue can be overloaded temporarily and makes the token bucket negative. This behavior gets amplified in real data compared with IMIX because the average negative value for a token bucket, average packet size, and average queue occupancy are larger than those for IMIX.

Table XIII. Compare Internet MIX with Real Packet Data (Number of Queues = 460000, $\alpha = 3000$, Interpacket Arrival Time = Constant, Simulation Time = 1s)

Statistic	IMIX	Chicago Dir A
Output queue occupancy	529.3	775.0
Average negative value for token bucket	−312.3	−383.6
Average packet size (bytes)	515	642

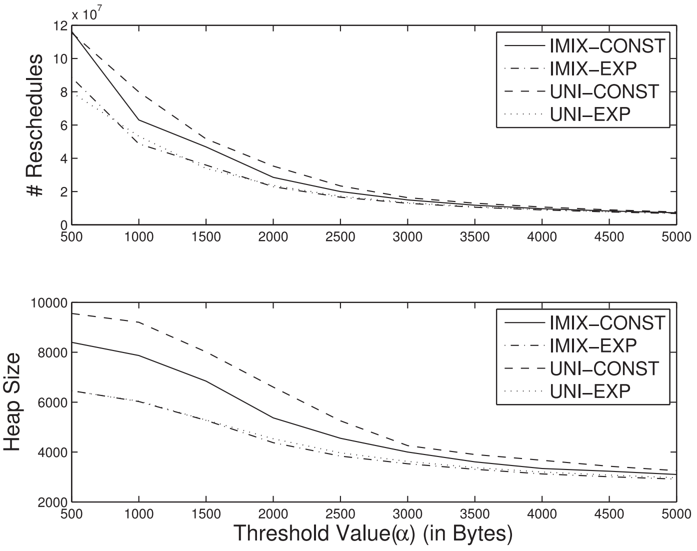


Fig. 12. Number of reschedule generated per queue and size of a heap per core with respect to different threshold values under severe congestion (overload = 1.29).

Using the information from real traces, we also measure the average packet delays to compare the proposed scheme with the conventional periodic fill scheme (token refill period = 1ms). In this test, we set the overload factor to 0.9˜0.95 so that the long-term average input bandwidth is smaller than the output bandwidth, which is a realistic setting in real routers/switches to guarantee reasonable packet delays. However, short-term overload happens due to back-to-back large packets under exponentially distributed interpacket delays and uneven distribution of packets over different queues. The simulation shows that average packet delays are 0.861ms (overload = 0.9) and 3.05ms (overload = 0.95) for the proposed scheme and 1.06ms (overload = 0.9) and 3.60ms (overload = 0.95) for the conventional periodic fill scheme. Considering the large number of slow queues (e.g., 460,000 queues) consisting of best-effort IP traffic, the refill period of 1ms for the periodic fill scheme fills the token slowly and causes slightly larger average delays than the proposed scheme.

5.3. Parameter Optimization: Number of Reschedules and Size of a Heap with Respect to the Reschedule Threshold (α)

In this experiment, we sweep the reschedule threshold value α to see its impact on the number of reschedules and the size of a heap. The upper plot of Figure 12 shows the number of reschedules generated per queue for 10s (overload factor is set to 1.29) for different reschedule threshold values from 500B to 4500B. The lower plot shows the average heap size per core during this simulation. The number of reschedules monotonically decreases as we increase the reschedule threshold α because the higher reschedule threshold increases the time for a queue to exhaust all tokens and reduces the reschedule frequency. Similarly, the heap size decreases with increasing α. A heap size is determined by the ratio between the schedulable and unschedulable time of a queue. If the schedulable periods of queues get bigger with respect to the unschedulable periods, the heap size is reduced. When a reschedule threshold increases, it increases schedulable periods because it takes longer to consume all tokens.

5.4. Cost of Proposed Method

We evaluate additional complexity due to the use of parallel heaps. The complexity of a software method can be assessed in terms of memory space used by the major data structure. Memory space needed in the conventional periodic fill method includes storage space for the token bucket, bandwidth, maximum burstiness per queue, and last update time. In the proposed scheme, we need a data structure to store a queue number and a reschedule time per heap element. In addition, we need one reschedule counter, for example, 4b per queue, which can be integrated with other preexisting per-queue data structure.

6. CONCLUSION AND FUTURE WORK

We propose a scalable software-based token bucket management scheme that can reduce memory accesses and power consumption significantly. To process tens of millions of token fill events per second under real-time and low-cost constraints, we propose novel parallelized heap data structures running on a manycore-based network processor. In addition, we quantitatively analyze the performance and memory footprint of the proposed software scheme using stochastic modeling. More specifically, we use the Lyapunov central limit theorem to accurately estimate the performance and cost of processing a large number of events to predict the behavior of a system under various different operating conditions. This stochastic modeling technique can be applied to model resource access behavior in manycore-based processor design. We have shown that the proposed scheme reduces the memory accesses by up to three orders of magnitude compared with periodic fills and effectively works with severely congested queues using the proposed adaptive reschedule threshold management scheme.

We are currently extending the proposed scheme to build compressed heaps for better memory footprints and studying trade-offs between processing power and memory usage. In addition, we plan to enhance the proposed scheme with various load balancing schemes based on the prediction of overloaded periods of queues.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

REFERENCES

A. Aeron. 2010. Fine tuning of fuzzy token bucket scheme for congestion control in high speed networks. In *2nd International Conference on Computer Engineering and Applications (ICCEA'10),* Vol. 1. IEEE, 170–174.

Agilent. 2007. The Journal of Internet Test Methodologies. Retrieved April 14, 2017 from https://intl.ixiacom.com/sites/default/files/resources/test-plan/agilent_ journal_of_internet_test_methodologies.pdf.

Salman A. AlQahtani. 2015. Token bucket fair scheduling algorithm with adaptive rate allocations for heterogeneous wireless networks. *Wireless Personal Communications* 84, 2, 801–819.

K. J. Åström and B. Wittenmark. 2013. *Adaptive control* (2nd. ed.). Courier Corporation, North Chelmsford, MA, US.

N. Beheshti, E. Burmeister, Y. Ganjali, J. E. Bowers, D. J. Blumenthal, and N. McKeown. 2010. Optical packet buffers for backbone internet routers. *IEEE/ACM Transactions on Networking* 18, 5, 1599–1609. DOI:http://dx.doi.org/10.1109/TNET.2010.2048924

J. C. R. Bennett and H. Zhang. 1996. WF 2 Q: Worst-case fair weighted fair queueing. In *INFOCOM'96. Proceedings of the 15th Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation.* Vol. 1. IEEE, 120–128.

P. Billingsley. 1995. *Probability and measure* (3rd. ed.). John Wiley & Sons, New York, NY.

N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2, 1–7. DOI:http://dx.doi.org/10.1145/2024716.2024718

The CAIDA. 2008. Statistical information for the CAIDA Anonymized Internet Traces. Retrieved April 14, 2017 from http://www.caida.org/data/passive/passive_trace_statistics.xml.

M. Campoy, A. P. Ivars, and J. Busquets-Mataix. 2001. Static use of locking caches in multitask preemptive real-time systems. In *Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*.

Veena S. Chakravarthi and M. Shilpa. 2013. Ingress flow based triple token bucket traffic control system for distributed networks. In *Proceedings of International Conference on VLSI, Communication, Advanced Devices, Signals & Systems and Networking (VCASAN-2013)*. Springer, 435–441.

Cisco. 2011. Cisco asr 9000 series ethernet line cards. Retrieved April 14, 2017 from http://www.cisco.com/en/US/prod/collateral/routers/ps9853/data_sheet_c78- 501338.pdf.

Cisco. 2014. The Cisco flow processor: Cisco's next generation network processor. Retrieved April 14, 2017 from http://www.cisco.com/c/en/us/products/collateral/routers/asr-1000-series-aggregation-services-routers/solutionoverviewc22-448936.pdf.

R. Ennals, R. Sharp, and A. Mycroft. 2005. Task partitioning for multi-core network processors. In *Compiler construction*. Springer, 76–90.

M. A. Franklin, P. Crowley, H. Hadimioglu, and P. Z. Onufryk. 2003. *Network Processor Design, Volume 2: Issues and Practices*. Morgan Kaufmann, San Francisco, CA.

R. Giladi. 2008. *Network processors: architecture, programming, and implementation*. Morgan Kaufmann, Burlington, MA.

S. Han, K. Jang, K. Park, and S. Moon. 2010. Packetshader: A GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review* 40, 4, 195–206. DOI:http://dx.doi.org/10.1145/1851275.1851207

E. Horowits, S. Sahani, and S. Anderson-Freed. 1992. *Fundamentals of data structures in c*. Computer Science Press.

X. Huang and T. Wolf. 2008. Evaluating dynamic task mapping in network processor runtime systems. *IEEE Transactions on Parallel and Distributed Systems* 19, 8, 1086–1098. DOI:http://dx.doi.org/10.1109/TPDS.2007.70806

Intel. 2005. Intel IXP2800 and IXP2850 network processors. Retrieved April 14, 2017 from http://int.xscale-freak.com/XSDoc/IXP2xxx/27853715.pdf.

H. Jeon, W. H. Lee, and S. W. Chung. 2010. Load unbalancing strategy for multicore embedded processors. *IEEE Transactions on Computers* 59, 10, 1434–1440. DOI:http://dx.doi.org/10.1109/TC.2009.181

V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim. 2013. EyeQ: Practical network performance isolation at the edge. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*. 297–311.

W. Kang, S. H. Son, and J. A. Stankovic. 2012. Design, implementation, and evaluation of a QoS-aware real-time embedded database. *IEEE Transactions on Computers* 61, 1, 45–59. DOI:http://dx.doi.org/10.1109/TC.2010.240

J. Kidambi, D. Ghosal, and B. Mukherjee. 2000. Dynamic token bucket (DTB): A fair bandwidth allocation algorithm for high-speed networks. *Journal of High Speed Networks* 9, 2, 67–87.

M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News* 33, 4, 92–99. DOI:http://dx.doi.org/10.1145/1105734.1105747

C. Networks. 2013. OCTEON III CN78XX Multi-Core MIPS64 Processors. Retrieved April 14, 2017 from http://www.cavium.com/pdfFiles/CN78XXPBRev1.0.pdf?x=1.

A. K. Parekh and R. G. Gallager. 1993. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking* 1, 3, 344–357.

E.-C. Park and C.-H. Choi. 2003. Adaptive token bucket algorithm for fair bandwidth allocation in diffserv networks. In *IEEE Global Telecommunications Conference (GLOBECOM'03)*. Vol. 6. IEEE, 3176–3180. DOI:http://dx.doi.org/10.1109/GLOCOM.2003.1258822

L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos. 2013. ElasticSwitch: Practical work-conserving bandwidth guarantees for cloud computing. *ACM SIGCOMM Computer Communication Review* 43, 4, 351–362.

I. Puaut and D. Decotigny. 2002. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Real-Time Systems Symposium (RTSS'02)*. IEEE, 114–123. DOI:http://dx.doi.org/10.1109/REAL.2002.1181567

S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. 2014. SENIC: Scalable NIC for end-host rate limiting. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. 475–488.

P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters* 10, 1, 16–19.

M. Shreedhar and G. Varghese. 1995. Efficient fair queueing using deficit round robin. In *ACM SIGCOMM Computer Communication Review*, Vol. 25. ACM, 231–242.

P. P. Tang and T.-Y. Tai. 1999. Network traffic characterization using token bucket model. In *INFOCOM'99. Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, Vol. 1. IEEE, 51–62. DOI:http://dx.doi.org/10.1109/INFCOM.1999.749252

Tilera. 2016. NPS-400: 400 Gbps NPU for Smart Networks. Retrieved April 14, 2017 from http://www.mellanox.com/related-docs/prodnpu/PBNPS-400.pdf.

G. Varghese. 2010. *Network algorithmics*. Chapman & Hall/CRC, Boca Raton, FL.

X. Vera, B. Lisper, and J. Xue. 2003. Data cache locking for higher program predictability. *ACM SIGMETRICS Performance Evaluation Review* 31, 1, 272–282. DOI:http://dx.doi.org/10.1145/885651.781062

R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic. 2002. Controlware: A middleware architecture for feedback control of software performance. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*. IEEE, 301–310. DOI:http://dx.doi.org/10.1109/ICDCS.2002.1022267

Received February 2016; revised November 2016; accepted January 2017