Supporting Multi-dimensional and Arbitrary Numbers of Ranks for Software Packet Scheduling

Jiaqi Zheng, Yanan Jiang, Bingchuan Tian, Huaping Zhou, Chen Tian, Guihai Chen, Wanchun Dou
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

***Abstract*—Compared with hardware implementation, the software packet scheduler uses the packet queuing data structure and a ranking function according to different dimensions to flexibly determine the packet dequeue order, which can significantly shorten the renewal cycles and increase the function deployment flexibility. The key data structure in prior work either bounds the number of rank or suffers from high computation overhead. In addition, they only support a single dimension and do not scale well. In this paper, we present Proteus, a software packet scheduling system that supports multi-dimensional and arbitrary numbers of ranks. We design a k-dimension heap data structure and develop "push" and "pop" algorithms to perform "enqueue" and "dequeue" operations. Furthermore, we implement a prototype of Proteus in software switch. Extensive experiments on BESS and numerical simulations show that Proteus can decrease the computation overhead, save the storage space and run much faster than state of the art.**

I. Introduction

Packet scheduling plays an important role on improving the application performance. For example, some network applications are latency-sensitive [1], [2], some of them are deadline-sensitive [3], [4] and others impose explicit constraints on the flow completion time [5], [6]. Packet scheduling [7]–[10] can provide differentiated services and optimize the service-level objective *e.g.*, minimizing the flow completion time, minimizing the deadline missing ratio, *etc.* According to a predefined dimension and the corresponding ranking function, the enqueued packets can be reordered based on the queuing data structure (*i.e.*, binary tree, AVL tree or queue, *etc.*). This procedure involves in a specific scheduling policy. When packets are dequeued, the scheduler can keep the current packet order or use a new dimension and ranking function to reorder the remaining packets in the queuing data structure.

Tab. I summarizes state-of-the-art hardware and software packet schedulers. Although the performance of software implementation like Eiffel [13] may be not as fast as that with hardware implementations such as pFabric [11] and PIFO [12], the software packet scheduling still presents its advantages. Say a commercial switch with eight physical queues per port [14], [15], these queues indicate different priorities, ranging from low to high. Once a packet arrives, it should be mapped to one of priority queues so that the switch can provide appropriate services. However, the possible number of the priorities in the packet header may be greater than that of priority queues. The hardware packet scheduler cannot implement more finer-grained classifications, especially for large number of applications in data centers [11], [16]. The short renewal cycles and deployment flexibility makes the software packet scheduler to replace hardware scheduler [11], [12] a promising approach.

Prior work such as pFabric [11] can only schedule packets according to one ranking function [12], [13], [17]. The rank value can be set to be the ToS (type of service) of a packet, enqueue time or others. Say a schedule requirement is that the packets with the highest ToS should be sent out firstly. However, there may be more than one packet sharing the same ToS. At this point, the scheduler requires sending the packet with the earliest deadline, which indicates that we want to schedule the most urgent one among the packets that share the same ToS. This requirement cannot be satisfied by pFabric since it only supports one dimension. Different from pFabric, the implementation of PIFO [12] relies on a priority queue, where the enqueued packet can be pushed into an arbitrary position and the dequeued packet can be popped from the head. PIFO can support arbitrary number of ranks, but it cannot reorder the buffered packets. Eiffel [13] extends the function of PIFO and adopts a fixed number of buckets as its data structure. The buffered packets among different buckets can be reordered and those in one bucket follow in a FIFO fashion. Although the ranking function of Eiffel for enqueue and dequeue operations can be two different ones, it cannot support arbitrary number of ranks.

In this paper we present Proteus, a software packet scheduling system that supports multi-dimensional and arbitrary numbers of ranks. We make three novel contributions in designing Proteus. First, we design a novel data structure — k-dimension heap — to support multi-dimensional and arbitrary numbers of ranks, where k is the number of dimensions. Based on this data structure, we develop "push" and "pop" algorithms to maintain the heap's property (*i.e.* reorder the buffered packets) during the packet queuing procedure. We prove that the time complexity of our algorithms is $\mathcal{O}(k \log n)$ and $\mathcal{O}(k \cdot (n + \log n)$, respectively, in the worst case, where k is the number of dimensions and n is the number of packets.

Our second contribution is a set of algorithms to apply k-dimension heap to the packet queuing system. Based on the k-dimension heap data structure, we fix one dimension as the enqueue time, where it stores the enqueued packet according to the order of arrival time. We modify the k-dimension heap data structure such that its corresponding algorithm can fit the ordered key index. Here delete the complexity.

978-1-7281-6887-6/20/$31.00 ©2020 IEEE.

978-1-7281-6887-6/20/$31.00 ©2020 IEEE

TABLE I
STATE-OF-THE-ART PACKET SCHEDULER COMPARISONS.

Scheduler	HW /SW	Data structure	Efficiency (enqueue)	Efficiency (dequeue)	Arbitrary numbers of ranks	Change the order of buffered packets	Multiple dimensions
pFabric [11]	HW	Push packet at the tail of the queue Pop packet at an arbitrary position	$\mathcal{O}(1)$	$\mathcal{O}(n)$	Yes	Support for each packet	No
PIFO [12]	HW	Push packet at an arbitrary position Pop packet at the head of the queue	$\mathcal{O}(n)$	$\mathcal{O}(1)$	Yes	No	No
Eiffel [13]	SW	One priority queue with N buckets	$\mathcal{O}(1)$	$\mathcal{O}(1)$	No	Support between buckets	No
Proteus	SW	k-dimension heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	Yes	Support for each packet	Yes

Our third contribution is a concrete implementation and evaluation of Proteus. We develop a prototype of Proteus based on the software switch [18]. We evaluate the Proteus prototype using BESS [19] with continuous traffic. We also conduct extensive simulations especially for incast scenarios using 15 senders and one receiver to evaluate our algorithms. Extensive experiments show that Proteus can decrease the computation overhead, save the storage space and run much faster than state-of-the-art.

II. RELATED WORK

We briefly review prior art on supporting multi-dimensional and arbitrary numbers of ranks.

Arbitrary number of ranks: Supporting arbitrary number of ranks requires the scheduler to provide differentiated service for any packets with different ranks. Originally, the first three bits of the ToS field [20]–[23] in the IP header were defined as IP precedence, which can support at most eight different services. Later, the first six bits of ToS was redefined as the DSCP (differentiated services code point) field in RFC 2474 [23]. The DSCP value can range from 0 to 63 and support 64 different services. However, the switches typically have eight priority queues and cannot provide more fine-grained packet scheduling [24]–[26]. pFabric [11] proposed to use two queues to support arbitrary number of ranks. One is used for storing the actual packets and the other holds the dimension information like priorities of those packets.

Multiple dimensions: There are vast literatures on scheduling strategies in modern switches. SP (strict priority) is one of the strategies that can guarantee that the flow with the higher priority would be scheduled before that with the lower one. WRR (weight round robin) and WFQ (weight fair queuing) can schedule packets according to a weight [27]. Recent works only use a single rank function to determine the scheduling order. PIFO [12] provides a priority queue which permits packets to be pushed into an arbitrary position based on its computed rank and dequeues from head. Eiffel [13] uses a fixed number of buckets which are ordered in FIFO fashion. OpenQueue [17] allows the operators to specify packet buffering architecture and policies through the customized language. Although these works only use a single rank, the rank functions always are diverse due to different schedule strategies. The work in [28] divided these works as two categorizes: *priority queues* that is used to determine the order of packets and *calendar queues* that is used to determine the departure times. For those common scheduling strategies, the rank can be priority, deadline, remaining flow size or arrival timestamp, which depends on the specific requirements in switches. The combination of all these dimension values can provide more flexible schedule strategies. For example, we can schedule the packet with the earliest deadline among the packets with the same priority. Furthermore, to solve the problem of sorting packets among multiple dimensions, the work in [29] proposed a data structure K-D heap that can achieve an efficient multi-dimensional priority queue. In a K-D heap, nodes at level i has the smallest key (*i.e.*, $key_{mod(i-1,K)+1}$) in its own subtree. In order to determine the smallest key_i, it needs to compare all the nodes in the highest i levels. For example, there are seven packets in Fig. 1 where the first yellow field indicates the packet priority, the second blue field indicates the timestamp, and the third red field indicates the deadline. These seven packets are organized as a 3-D heap. The root node has the highest priority; the nodes in the second level have the smallest timestamp and the nodes in the third level have the earliest deadline in each subtree. To determine the packet with the highest priority, we can first get the root node. However, if we plan to find the packet with the smallest time sequence or deadline, we need compare $2^K - 1$ nodes at most. Furthermore, many packets will share the same key like priority. In this case, the scheduler needs another dimension information to make an accurate decision. If the scheduler wants to send the packet with the earliest deadline among the packets with the same priority, then the packet $(3, 17, 69)$ should be scheduled first. In a K-D heap, the nodes with the same key value can be located in different levels, which makes an efficient schedule hard to design. Also the insertion and deletion operations are very frequent during the scheduling procedure. The most important thing is to determine the packet with the smallest or largest key value at each time. Hence,

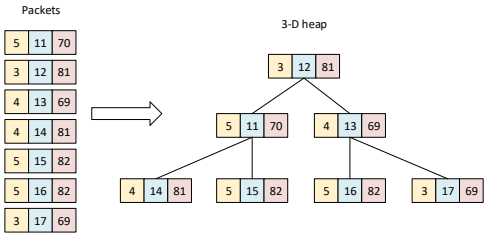


Fig. 1. Packets are organized as K-D heap (K = 3). All packets have three dimensions: priority, enqueue time sequence and deadline, which can be represented in yellow field (key_1), blue field (key_2) and red field (key_3), respectively. In this 3-D heap, the node (3, 12, 81) in the first level has the smallest key_1 among its descendants; the nodes (5, 11, 70) and (4, 13, 69) in the second level have the smallest key_2 among their descendants respectively; the nodes in the third level should have the smallest key_3 among their descendants if they are not leaves.

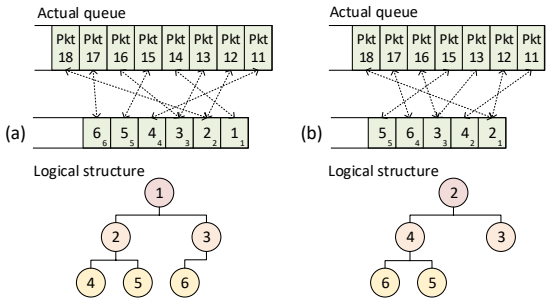


Fig. 2. Comparison using k-dimension heap, where k equals one and this dimension represents the priority. The schedule policy is that the packets with high priority (the smaller number) will be send out firstly.

sorting all packets incurs more time overhead and thus is not necessary [30].

Our work is complementary to previous work. The novelty lies in a novel data structure that supports multi-dimensional and arbitrary numbers of ranks, which can support more flexible strategies. This novel data structure enables the operators to specify multiple dimensions as their rank functions and can be integrated to OpenQueue to make the management more powerful.

III. Proteus Overview

Traditional switches with 8 priority queues are not enough to support arbitrary number of ranks. Creating more fixed number of priority queues is not a scalable choice since the total number of different ranks for scheduling cannot be a prior. The simplest implementation is to put all packets into a single queue and rely on scheduling policies to differentiate the ranks like pFabric and PIFO. Since all the packets are buffered in a single queue, dimensions such as priority, deadline, flow size, and waiting time *etc.* are collected by the switch to make a dequeue decision. For example, the packets with the highest priority should be sent out first. If two packets carry the same priority, the packet with earlier arrival time will be sent out. This case requires two dimensions, one for priority and the other for arrival time. In particular, we develop a new data structure — k-dimension heap — to support arbitrary number of ranks and multiple dimensions.

We use the example of k-dimension heap shown in Fig. 2 to illustrate how Proteus works, where k equals one, *i.e.* there are only one dimension and the packets dequeue operation only depends on the priority. In our example, when a new packet arrives, the packet will be pushed back at the tail of the packet queue Then the rank value (here is the priority) will be added into the heap. For dequeue operation in Fig. 2(a), the packet with time sequence "14" has the highest priority "1" and can be directly determined to dequeue. After the heap re-adjusting procedure, the data structure is shown in Fig. 2(b). Then the packet with time sequence "12" and the priority "2" is selected to dequeue. The duplicate data (the node numbered as priority "2" and "3") in the heap can be only stored once in Fig. 2. We use pointers to direct corresponding data packets, which can avoid redundant comparisons. For more dimensions, there should be an individual heap for each dimension so that Proteus can efficiently pick the smallest key value directly.

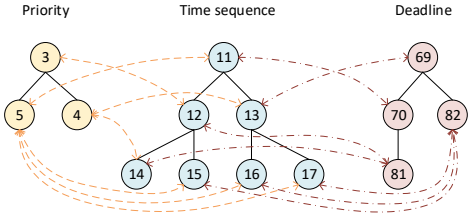


Fig. 3. Illustration of a 3-dimension heap. There are three dimensions: priority, deadline and time sequence.

The central challenge in designing Proteus is how to design a data structure to maintain the dynamic relationships among each individual heap and the corresponding algorithms to support multi-dimensional and arbitrary numbers of ranks, which is our focus in the following sections.

IV. Proteus Design

In this section, we first design a new data structure — k-dimension heap — to support multi-dimensional and arbitrary numbers of ranks, where k is the number of dimensions. And we further propose a set of algorithms and analyze their time complexity. Finally, we apply the proposed algorithms to the queue.

A. Basic structure

The k-dimension heap is a generalization of the classic heap, which includes k subheaps and each subheap can represent one dimension. There is at least one subheap (dimension) selected as the primary subheap, where the nodes have one-to-one mappings with packets. Without loss of generality, we use the "time sequence" (the enqueue timestamp) as the nodes in the primary subheap as it can uniquely identify the packets. Hence, the number of nodes in the primary subheap is equal to the number of packets in the queue, while may not hold in other subheaps due to duplicate elements. For example, many packets have the same priority and thus the number of nodes in the subheap with priority type may be less than the number of packets. The dimensions (*i.e.* priority, deadline, time sequence, *etc.*) of a packet are distributed to different subheaps and connected by a set of dashed lines. Each node in the primary subheap records the location of all relevant nodes in other subheaps (dimensions). The node in the non-primary subheap only keeps a pointer vector to record the relationship between the primary subheap and the non-primary subheap.

Let us first introduce the following definition and property.

Property IV.1. *Each subheap is a binary tree, for each node in which, its value is unique and should be less than that of its left child and right child.*

Property IV.2. *Each node represents a value corresponding to one dimension. If two nodes in two subheaps are connected by a dashed line, then the corresponding values of different subheaps belong to the same packet.*

Definition IV.1. The k-dimension heap *is composed of k subheaps, where the property of each subheap satisfies*

Property IV.1. There is a unique primary subheap with one-to-one mapping for real packets, the relationship among subheaps and primary heap satisfies Property IV.2.

Taking Fig. 3 as an example, we can see that it is a 3-dimension heap, where the nodes in each subheap represent the time sequence, priority and deadline, respectively. Note that the time sequence forms a one-to-one mapping to the nodes and thus it is a primary subheap. Specifically, the node with time sequence "11" has the priority "5" and deadline "70". The dashed lines are established to capture their relationships. The k-dimension heap structure supports arbitrary number of ranks and multiple dimensions, satisfying the diverse requirements of using different scheduling strategies. Note that a single heap storing ordered tuples (*i.e.* priority, deadline, time sequence, *etc.*) cannot work since the order varies with the different dimensions. Next we introduce the "push" and "pop" operations for k-dimension heap, which are summarized in Algorithm 1 and Algorithm 2, respectively.

The "push" operation. When a new packet arrives, the switch obtains the priority, deadline and other dimension information and assigns a unique sequence number to this packet. Accordingly a new node with this sequence number will be created and added to the primary subheap. Other subheaps do not need to create a new node if the value in this dimension already exists. Finally we need to connect the nodes in subheaps and the primary heap. The entire procedure for the "push" operation is shown in Algorithm 1. We denote $h[p]$ as the p_{th} subheap, where $p = 1, 2, \cdots, k$. Accordingly, we can access the i_{th} node in the p_{th} subheap by $h[p][i]$ directly. Without loss of generality, we set the first heap (*i.e.*, $h[1]$) as the primary subheap. Each node in the subheap stores a value for a specific dimension and a pointer vector that is used to record other relevant nodes. Specifically, the pointer vector in the nodes of the primary subheap keeps k pointers, where the j_{th} pointer directs to the relevant node in $h[j]$. For example, in Fig. 3, we assume that the time sequence subheap is $h[1]$, the priority subheap is $h[2]$ and the deadline subheap is $h[3]$. The node in time sequence subheap $h[1]$ maintains the pointer vector with three elements. The first one can be a null pointer. The second pointer points to the node with value 5 in the priority subheap, and the third one points to the node with value 70 in the deadline subheap. As for the nodes in $h[2]$ or $h[3]$, they only record the pointers that direct to $h[1]$, where the size of the pointer vector is dynamic. In the priority subheap, the node with value 4 has two pointers in its pointer vector, and the node with value 5 has four. On a high level, the switch first obtains the k values corresponding to k dimensions for the new arrival packet and puts them into a vector. Then the newly created node $nodePtr[1]$ as the first element is pushed to the tail of the primary subheap $h[1]$. For all other subheaps $h[p]$ ($p \neq 1$), the value of the new node could be duplicated with existing nodes. If this condition holds, we cannot add this new node $nodePtr[p]$ into the subheap. Otherwise, we push it to the tail of the subheap, recorded as $nodePtr[p]$ and readjust the subheap to satisfy Property IV.1. Finally, we update the pointer vector of both $nodePtr[1]$ and $nodePtr[p]$ to maintain the relationship among subheaps.

Algorithm 1: The push operation in k-dimension heap

Input: The vector v with k elements, each element represents the value of a dimension.

1 $n \leftarrow n + 1$;
2 $nodePtr[1] \leftarrow createNode(v[1], k)$;
3 $h[1].push\_back(nodePtr[1])$;
4 $i \leftarrow n$;
5 **while** $i \neq 1$ *and* $h[1][i].value < h[1][\lfloor i/2 \rfloor].value$ **do**
6 $\quad swap(h[1][i], h[1][\lfloor i/2 \rfloor])$;
7 $\quad i \leftarrow \lfloor i/2 \rfloor$;
8 **for** $p = 2$ *to* k **do**
9 $\quad nodePtr[p] \leftarrow h[p].getNodePtr(v[p])$;
10 $\quad$ **if** $nodePtr[p]$ *is* $NULL$ **then**
11 $\quad\quad nodePtr[p] \leftarrow createNode(v[p], 0)$;
12 $\quad\quad h[p].push\_back(nodePtr[p])$;
13 $\quad\quad i \leftarrow h[p].size()$;
14 $\quad\quad$ **while** $i \neq 1$ *and* $h[p][i].value < h[p][\lfloor i/2 \rfloor].value$ **do**
15 $\quad\quad\quad swap(h[p][i], h[p][\lfloor i/2 \rfloor])$;
16 $\quad\quad\quad i \leftarrow \lfloor i/2 \rfloor$;
17 $\quad nodePtr[p] \rightarrow recordVector.push\_back(nodePtr[1])$;
18 $\quad (nodePtr[1] \rightarrow recordVector[p]) \leftarrow nodePtr[p]$;

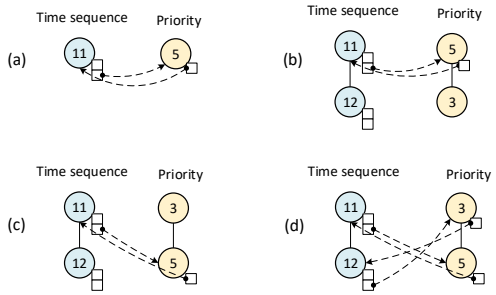


Fig. 4. Illustration of push operation in the 2-dimension heap where the first dimension is the time sequence and the second dimension is the priority. The vector information of the first packet is $\langle 11, 5\rangle$ and that of the second packet is $\langle 12, 3\rangle$. The subheap with the time sequence is the primary subheap.

At the beginning of Algorithm 1, the variable n records the total number of vectors stored in the k-dimension heap (*i.e.*, the number of nodes in the primary subheap $h[1]$ where its value is unique). For every push operation, the variable n increases by one (line 1). The pointer vector $nodePtr$ records the node information from vector v for each dimension. In the primary subheap $h[1]$, since its value is unique, we can always

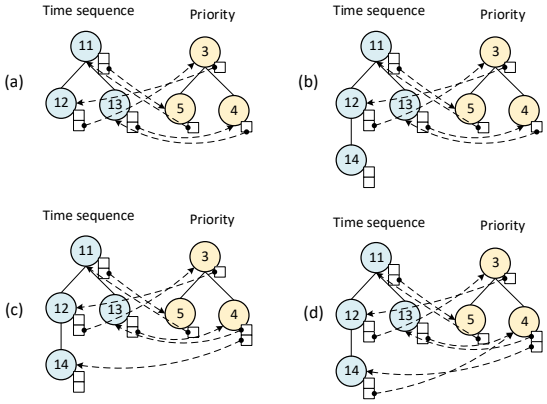


Fig. 5. Illustration of push operation in the 2-dimension heap where the first dimension is time sequence and the second dimension is priority. The vector information of the first packet is $\langle 11, 5\rangle$, that of the second packet is $\langle 12, 3\rangle$, that of the third packet is $\langle 13, 4\rangle$ and that of the fourth packet is $\langle 14, 4\rangle$. The subheap with time sequence is the primary subheap.

create a new node from the information of $v[1]$ (line 2) and push the new node to the tail of the $h[1]$ (line 3). After a new node is added into the subheap $h[1]$, we need to readjust this subheap (lines 4-7). For each non-primary subheap $h[p]$, we need to check whether the value of the new node exists or not (line 9). If the value of the new node does not exist, we can do the same procedure as in the primary subheap $h[1]$ (lines 10-16). Otherwise, we do not need to add a new node and just record the pointer of the existing node in $nodePtr[p]$ directly. Finally, the pointer $nodePtr[p]$ will be added to record the relationship to $nodePtr[1]$ in $h[1]$ (line 17). At the same time, the pointer $nodePtr[1]$ in the primary subheap updates the p_{th} pointer to establish the relationship to the pointer $nodePtr[p]$ in the non-primary subheap (line 18).

Now using the running example as shown in Fig. 4 and Fig. 5, we illustrate how the "push" operation in Algorithm 1 works. Initially, each subheap contains only one node, and the nodes keep the pointers of each other, as shown in Fig. 4(a). When the second packet arrives, the switch obtains its priority with value "3" and assigns a sequence number with value "12". For the primary subheap, we create a new node with value "12" and the size of the pointer vector is 2 ($k = 2$). We push the new node at the tail of the primary subheap, *i.e.* time sequence subheap. There is no need to change its position since value "12" is larger than value "11". For the priority subheap, there is only one node with value "5" and the value "3" is not a duplicate element. Hence we create a new node with value "3" and an empty pointer vector, pushing it into the tail of the priority subheap as in Fig. 4(b). Since value "3" is smaller than value "5", we need to exchange their positions to satisfy Property IV.1. Accordingly, the priority subheap should be readjusted from Fig. 4(b) to Fig. 4(c). Finally given the pointer of the new nodes, we establish the relationship between the node with value "12" in the time sequence subheap and the node with value value "3" in the priority subheap, as shown in Fig. 4(d).

Fig. 5 shows a special case that the new node shares a value with an existing node. The vector information of the fourth packet is $\langle 14, 4\rangle$, which indicates that the value "4" has existed in the priority subheap and we can only add the value "14" into the time sequence subheap as shown in Fig. 5(b). Since the node with value "4" already exists in the priority subheap, we only need to update the pointer of the node with value "4" to establish the relationship to the new node in the time sequence subheap. Finally, we update the pointer vector as shown in Fig. 5(d). For more dimensions, since the non-primary subheaps only connect with the primary subheap, the operations of other non-primary subheaps can be the same as that of the priority heap in our example.

The "pop" operation. When the switch performs the dequeue operation, its decision is based on one or more dimensions. For example, the packet with higher priority is usually sent out first. If the priority of two packets is identical, the packet with earlier arrival time is sent out first. We use m, m', m'', $\cdots$ to denote the first, second, third, $\cdots$ dimension in the switch's decision. However, one node in a subheap $h[m]$ may map several nodes in the primary subheap since the node value in the non-primary subheap may not be unique. This indicates that we cannot remove a node in the non-primary subheap if it has more than one pointer. In general, we first obtain the pointer of node in the primary subheap to be deleted and determine the index $r[1]$ in its subheap. We exchange the $r[1]_{th}$ node with the last one in the primary subheap and readjust the heap structure. For the other subheaps, we determine the relevant nodes via the pointer vector of the node in the primary subheap and the position $r[p]$ of the p_{th} subheap is recorded. We check whether the node pointer in the primary subheap is the only one of the $r[p]_{th}$ node or not. If this condition holds, we exchange the $r[p]_{th}$ node with the last one in the p_{th} subheap and readjust the heap from the $r[p]_{th}$ position. When its value is recorded, we delete the last one and free the space. Otherwise, we only remove the pointer from the pointer vector for the $r[p]_{th}$ node in the primary subheap. When the check operation for all non-primary subheaps is done, we delete the last one in the primary subheap.

At the beginning of Algorithm 2, the vector r records the index of the nodes in each subheap that needs to be popped and the $nodePtr$ records the pointer of the corresponding nodes. We first determine the node to be popped in the primary subheap as this node has all the pointers to the other subheaps. If the input dimension m is the primary subheap, the $nodePtr[1]$ and $r[1]$ can be set to be $h[1][1]$ and 1 directly (lines 2-3). Otherwise, the $nodePtr[1]$ can be determined by the dimension sequence $m, m', m'', \cdots$ step by step. The

Algorithm 2: The pop operation in k-dimension heap

Input: The sequence $m, m', m'', \cdots \in \{1, 2, \cdots, k\}$ which indicates the decision standard among dimensions.
Output: The vector v that needs to be popped.

1 **if** $m == 1$ **then**
2 $nodePtr[1] \leftarrow h[1][1]$;
3 $r[1] \leftarrow 1$;
4 **else**
5 $nodePtr[1] \leftarrow$ the node pointer in the primary subheap;
6 $r[1] \leftarrow getLocation(nodePtr[1])$;
7 $swap(h[1][r[1]], h[1][n])$;
8 $i \leftarrow r[1]$;
9 Readjust(i,1);
10 $v[1] \leftarrow h[1][n].value$;
11 **for** $p = 2 \rightarrow k$ **do**
12 $nodePtr[p] \leftarrow h[1][n].recordVector[p]$;
13 $r[p] \leftarrow getLocation(nodePtr[p])$;
14 $v[p] \leftarrow h[p][r[p]].value$;
15 **if** $h[p][r[p]].recordVector.size() > 1$ **then**
16 $removePtr(h[p][r[p]].recordVector, nodePtr[1])$;
17 **else**
18 $size[p] \leftarrow h[p].size()$;
19 $swap(h[p][r[p]], h[p][size[p]])$;
20 $i \leftarrow r[p]$;
21 Readjust(i,p);
22 $h[p].pop\_back()$;
23 $h[1][n].recordVector[p] \leftarrow null$;
24 $deleteNode(nodePtr[p])$;
25 $h[1].pop\_back()$;
26 $deleteNode(nodePtr[1])$;
27 $n \leftarrow n - 1$;
28 **return** v;

Algorithm 3: Readjust (index i, heap p)

Input: The index of node i and the heap that node i belongs to.

1 **repeat**
2 $t \leftarrow 2i$;
3 **if** $t+1 < size[p]$ *and* $h[p][t+1].value < h[p][t].value$ **then**
4 $t \leftarrow t+1$;
5 **if** $h[p][t].value < h[p][i].value$ **then**
6 $swap(h[p][t], h[p][i])$;
7 $i \leftarrow t$;
8 **until** $2i \geq size[p]$ *or* $i \neq t$;

value of $r[1]$ can be determined from the $nodePtr[1]$ (lines 5-6). Next in subheap $h[1]$, we need to exchange the node $nodePtr[1]$ with the tail of this subheap and readjust the subheap structure without the last one (lines 7-9). For the remaining non-primary subheaps, we obtain the node pointer $nodePtr[p]$ by the record information stored in $nodePtr[1]$ and then determine the index $r[p]$ (lines 12-13). For the nodes in the non-primary subheap, some of them point to more than one nodes in the primary subheap. In this case, the node $nodePtr[p]$ cannot be popped. We should only remove the pointer $nodePtr[1]$ from the recordVector (lines 15-16). As in the subheap $h[1]$, we exchange the node $nodePtr[p]$ with the tail node and readjust the subheap without the last one (lines 19-21). Since the value of $nodePtr[p]$ is already recorded (line 14), we remove the node in subheap $h[p]$ (line 22), delete the pointer of $nodePtr[p]$ to the node $nodePtr[1]$ in the primary subheap (line 23) and finally free the storage space of $nodePtr[p]$ (line 24). When all the nodes in the non-primary subheap complete the pop operations, we remove the node $nodePtr[1]$ in subheap $h[1]$ and free the storage space (lines 25-26). Accordingly, the variable n should be decreased by one (line 27).

Now we use the running example of Fig. 6 and Fig. 7 to illustrate how the "pop" operation in Algorithm 2 works. We assume that the switch performs dequeue operation by two dimensions — priority and time sequence, *i.e.* the packet with high priority will be sent out first. If two packets have an identical priority, the packet with earlier arrival time (smaller time sequence number) will be sent out first. The initial heap structure is shown in Fig. 6(a), which has the same setting as that of Fig. 5(d). Here the switch needs to perform dequeue operation to determine the packet with priority "3" and time sequence "12". The pointer of node with value "3" in the priority subheap and that of the node with value "12" in the time sequence subheap points to each other and no other pointers point to them. Hence we can remove these two nodes directly. We first obtain the node with value "12" in the time sequence subheap (primary subheap) via the pointer of node with value "3" in the priority subheap. Once the node with value "12" in the time sequence subheap is determined, we exchange the second node with the last node (*i.e.* the node with value "12" and the node with value "14") in the time sequence subheap. Accordingly, we record the index to be removed as $r[1]$. When the exchange operation in the time sequence subheap is done, the heap structure is shown in Fig. 6(b).

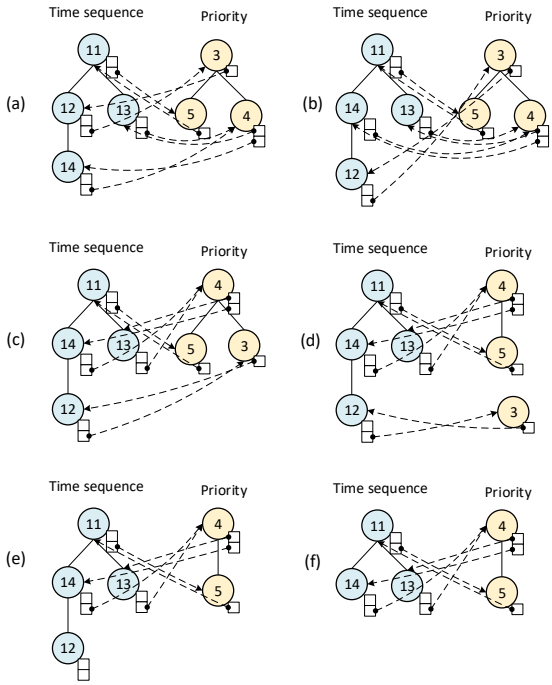


Fig. 6. Illustration of the pop operation in the 2-dimension heap where the first dimension is time sequence and the second dimension is the priority. The vector information to be popped is $\langle 12, 3\rangle$. The subheap with time sequence is the primary subheap.

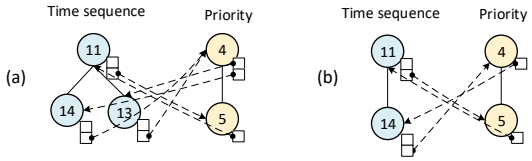


Fig. 7. Illustration of the pop operation in the 2-dimension heap where the first dimension is time sequence and the second dimension is the priority. The vector information to be popped is $\langle 13, 4\rangle$. The subheap with time sequence is the primary subheap.

For the priority subheap, the exchange operation between the node with value "3" and that with value "4" cannot violate the Property IV.1 as well. We record the index to be removed as $r[2]$ and the heap structure is shown in Fig. 6(c). Then we can pop the node out of the subheap as shown in Fig. 6(d). Next we clear the pointer recorded by the node in the priority subheap and free the occupied space (Fig. 6(e)). Finally, we pop the node in the time sequence subheap and free its occupied space. The final heap structure is shown in Fig. 6(f).

Fig. 7 shows another example when performing pop operation. Here the head node with value "4" in the priority subheap maps two nodes with value "13" and value "14" in the time sequence subheap in Fig. 7(a). According to the rules that the packet with the earlier arrival time (smaller time sequence number) will be sent out first, we obtain the first pointer of node with value "4" in the priority subheap and determine the node with value "13" in the time sequence subheap. The node with value "13" is already the last one in the time sequence subheap so that we do not need to readjust its structure. For the node with value "4" in the priority subheap, we cannot directly remove it since another mapping relationship exists (the node with value "14" in the time sequence subheap) Instead, we

only delete the pointer pointing to the node with value "13" in the time sequence subheap. Finally, we pop the node with value "13" in the time sequence subheap and free its space as well. When all is done, the heap structure is shown in Fig. 7(b). Note that for more dimensions, the operations of other non-primary subheaps perform the same as that of the priority heap before the node in the primary subheap is popped.

Theorem IV.1. *The time complexity of Algorithm 1 and Algorithm 2 are* $\mathcal{O}(k \cdot (n + \log n))$.

Proof. For a k-dimension heap, the total number of nodes in each subheap is at most n, where n is the number of the packets in the switch. In general, a push or pop operation in one subheap requires $\log n$ comparisons and swaps on the worst case. And all k subheaps need at most $k \log n$ times when all subheaps need to perform addition or deletion operation. In Algorithm 1, the time complexity largely depends on the function $getNodePtr$ (line 9) that may cause $\mathcal{O}(n)$ time complexity in our k-dimension heap. Note that this function needs to be called in all $k-1$ non-primary subheaps, and thus the total time complexity of Algorithm 1 is $\mathcal{O}(k \cdot (n + \log n))$.

For Algorithm 2, to determine a node to be deleted in the primary subheap, the worst case is $\mathcal{O}(n)$ time complexity. If we need multiple dimensions to make a decision together, we need to obtain the packet set that shares the smallest value for the first dimension m and check all of them. In practice, if the size of set is very large (*e.g.*, larger than $n/2$), we can use other dimensions $(m', m'', \cdots)$ to determine the packets and check whether it has the smallest value in dimension m to reduce computation overhead. When a node in the primary heap is deleted, we need to remove the corresponding pointer in the non-primary heap (line 16 in Algorithm 2). The corresponding node can be deleted only if the node in the primary heap and the corresponding node in the non-primary heap have one-to-one mappings. The procedure of determining to be removed point also needs $\mathcal{O}(n)$ time complexity. Hence, the total time complexity of Algorithm 2 is $\mathcal{O}(k \cdot (n + \log n))$ as well. □

B. Apply k-dimension heap to the queue in parallel

In this subsection, we discuss how to apply the k-dimension heap to the queue in the switch. It's intuitive to use the time sequence as the primary heap, as the sequence number is unique and increases by one once a new packet arrives. For this kind of dimension in the primary heap, we slightly modify the heap data structure to queue data structure since the value in the time sequence dimension is monotonically increasing. Fig. 8 and 10 presents two examples, where the primary subheap becomes a queue and other subheaps keep unchanged, a flag field added to each node in the primary subheap to indicate whether the node is available or not. A node is available means that the corresponding packet is still in the queue. In the modified k-dimension heap data structure, once obtaining a new dimension vector, the node in the primary dimension is simply added to the tail of the queue, where the flag field is set to be true. The nodes in other dimensions perform the same as that in the k-dimension heap. When performing pop operation, it should first determine the node in the primary subheap (queue), and obtain the node pointers of other dimensions. After the operations in all subheaps is done, we set the flag field to be false, which indicates that this vector has already logically removed. At the end of pop operations, we check each node in the primary subheap, and delete all the nodes whose flag fields are false at the head of the queue. Fig. 9 shows an example of popping the second packet in Fig. 8. The grey node with time sequence "12" in the queue will be removed when all previous nodes (the node with time sequence less than "12") are marked gray.

In the k-dimension heap structure, all the non-primary subheaps only connect with the primary subheap. It makes possible that the comparison and swap operations in different non-primary subheaps are computed in parallel. Taking the push operation of k-dimension heap as an example, while the nodes in the primary subheap have already been added, the nodes in other dimensions can be added or stored in parallel since the memory space for nodes and pointers that they access are different. The function $getNodePtr$ in "push" operation may cause $\mathcal{O}(n)$ time complexity, which is used to check whether the corresponding value exists or not. In practice, we can implement it in parallel. Specifically, we can compare the given value with the value in the root node firstly. If this value is smaller than that in the root node, then the pointer returns empty. If these two values are equal, it returns the pointer of the root node. Otherwise we compare this value with two subheaps. Checking the subheaps can be performed in parallel and searching range is at most $\log n$. Hence, Algorithm 1 can be finished in $\mathcal{O}(\log n)$ time complexity. In the pop operation, the computation in the non-primary subheaps can be done in parallel in a similar way. Since these subheaps are independent to each other both in logical (different subheaps) and physical spaces (different memory space), it is feasible to implement them with multi-thread programs.

V. Experimental Evaluation

We have conducted extensive experiments to evaluate our algorithms. In this section we report our performance evaluation using the software switch implementation and the NS3 simulations.

Benchmark Schemes: We compare the following schemes with our algorithms.

- **Proteus**: Our proposed algorithms using the k-dimension heap data structure.
- **AVL Tree** [31]: The buffered packets are organized by the AVL tree, *i.e.* the height difference of two subtrees is always less than or equal to one. If the difference is more than one due to an insertion or deletion, the AVL tree will perform rotation operations to keep this property.
- **RB Tree** [32]: The buffered packets are organized by the RB Tree. Each node in the red-black tree has a color — red or black, where the root node must be black. If a node is red, both children are black. Furthermore, every path from a given node to any of leaf nodes contains identical number of black nodes.

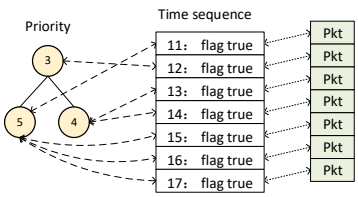


Fig. 8. Apply the 2-dimension heap to the queue.

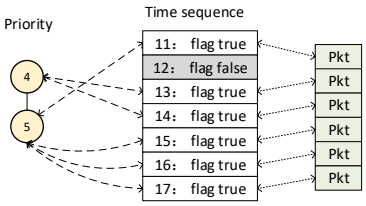


Fig. 9. The pop operation for the second packet in 2-dimension heap.

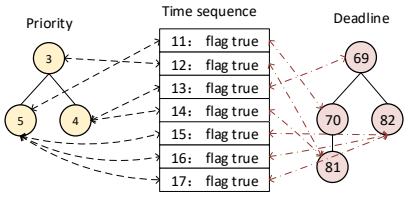


Fig. 10. Apply the 3-dimension heap to the queue. For simplicity, we do not show the structure storing the real packets

- **K-D Heap** [29]: The buffered packets are organized by K-D Heap that uses one single heap to implement a multi-dimensional priority queue.
- **pFabric** [11]: pFabric can push a packet to the tail of the queue and pop the packet at an arbitrary position.
- **PIFO** [33]: PIFO can push a packet at an arbitrary position and pop a packet at the head of the queue.
- **Eiffel** [13]: Eiffel uses a fix number of buckets to store the buffered packets, where each one is a FIFO queue and has a limited range of rank values.

A. Implementation and Testbed Emulations

Implementation: We implement Proteus in the software switch, where the time sequence is the primary subheap in the k-dimension heap data structure. We detail our implementation both in the host and the switch. **(i) The host TC layer.** Linux offers a rich set of functions to perform traffic control, that can determine which packet can be enqueued or dropped. The traffic control layer lies in the position between the NetDevices (L2) and network layer (L3). We implement the k-dimension heap in the traffic control layer (as shown in Fig. 11) and do not modify the net card code. Once a new packet arrives, the TC layer performs the enqueue operation. The function in the TC layer collects all dimension information as a vector for this new packet and pushes the dimension vector to the heap. For the dequeue operation, we obtain all index elements from the k-dimension heap as a vector by the sequence $m, m', m'', \cdots \in \{1, 2, \cdots, k\}$. We determine the corresponding packet in the TC layer via the vector v. According to the pre-defined schedule policy, the function in the TC layer can make the choice of sending the packet out of the net card or dropping it. **(ii) The switch.** The switch has limited memory and thus we need to avoid the redundant storage. It is obvious that the order of data packets is the same as that of time sequences. Here, we use the packet queue as the primary subheap in switch. Note that the node in the primary subheap will be deleted after performing dequeue operations, while the packet in the host cannot be deleted. The reason is that, besides the TC layer, the packet may be used in other layers. The k-dimension heap is located at each egress port of switch, as shown in Fig. 11. For the enqueue operations, when a new packet arrives, the switch collects the dimension information as a vector. Then we set the primary subheap as the packet itself. Finally we push the nodes into the k-dimension heap. For the dequeue operations, before removing a packet, we need an input sequence to determine which packet

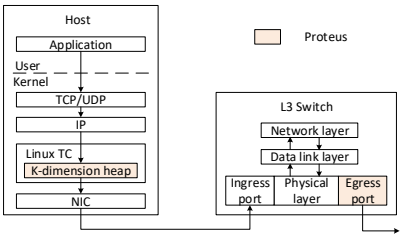


Fig. 11. Software stack of Proteus's implementation.

should be dequeued or dropped. We obtain the dimension information from the k-dimension heap according to the input sequence $m, m', m'', \cdots$ as the decision standard, and the pointer of the packet to be removed is the first element in the vector.

Testbed Setup: Our testbed contains one server which sends loopback data to test the performance of Proteus. The server is a DELL PowerEdge R730, equipped with a 24-core Intel Xeon E5-2650V4 CPU, 64GB RAM. The server runs Ubuntu Server 18.04. We implement Proteus, pFabric, PIPO and Eiffel based on the BESS [19] at the server, which is the state-of-the-art software switch implementation. We have a source to generate packets, which pass through the queue structure and then forward to a receiver. We measure the CPU consumption via observing the CPU cycles from this four data structures when the switch performs dequeue operations.

Experiment Results: We first investigate the CPU cycles variations using `monitor tc` command in Fig. 12. The CPU cycles consumption of Proteus is stable and slightly less than that of pFabric. The essential reason is that the search operations dominate the computation overhead, which heavily depend on the number of the buffered packets. In general, the more buffered packets, the more CPU cycles will be consumed. However, Proteus only compares the nodes of the parent and the child in the heap, and the search operations cannot be significantly affected by the number of arriving packets. The CPU cycles consumption of PIFO and Eiffel are relatively low for the dequeue operation since they do not require comparison operations for the buffered packets.

We evaluate the comparison times and the occupied storage space for Proteus , pFabric, PIFO, Eiffel. The maximum queue size in the switch is set to be 100 packets in this setting (Eiffel is set to be 100 buckets). The CDF of the comparison times for enqueue and dequeue operation are shown in Fig. 14(a) and Fig. 14(b). Suppose that there are n buffered packets in the switch, we can know that the enqueue operations for Proteus require at most $\log n$ times comparisons. pFab-

TABLE II
THE RUNNING TIME COMPARISONS FOR DIFFERENT NUMBERS OF BUFFERED PACKETS.

Different numbers of buffered packets		2000	4000	6000	8000	10000
Enqueue	Proteus	0.3	0.4	0.4	0.4	0.4
	K-D Heap	0.3	0.4	0.4	0.4	0.4
	AVL Tree	13.7	21.8	25.6	39.6	48.5
	RB Tree	8.2	17.1	22.4	38.0	40.0
Dequeue	Proteus	0.6	0.7	0.8	0.9	1.2
	K-D Heap	12.4	15.5	18.0	25.8	33.2
	AVL Tree	2.4	2.6	2.7	2.9	2.9
	RB Tree	1.0	1.0	1.1	1.2	1.2

ric does not need comparison operations and PIFO needs $n-1$ times comparison [33]. As for Eiffel, it only needs to lookup operations for a bucket. The dequeue operations for Proteus also requires at most $\log n$ times comparisons, while pFabric needs $n-1$ times, PIFO does not need comparison operations and Eiffel uses the bucket bitmap to get the packet. The occupied storage space of dimension data comparison is shown in Fig. 13. In this figure, Proteus can save the storage space by 80% compared with pFabric. The reason is that the duplicate elements in Proteus can only occupy one unit storage space using our heap structure. PIFO does need to store the dimension data, and Eiffel only needs to store the bitmap for only a few bytes.

B. Simulation

Setup: We have implemented all schemes in C/C++ and have simulated the enqueue and dequeue operations in NS3 to show the performance. Here we use 15 sending hosts with one receiving host, transmitting 100M data totally. The maximum queue size of each switch is set to be 1000 packets. We compare the running time, the consumed CPU cycles and the occupied storage space for enqueue and dequeue operations. Each point is at least an average of ten runs.

Experiment Results: We first investigate the running time for different schemes — Proteus, K-D heap, AVL Tree and RB Tree. The results are shown in Fig. 15(a) and Fig. 15(b) for enqueue and dequeue operations when we vary the number of dimensions k. For the enqueue operation in Fig. 15(a), we can observe that the total running time of Proteus, K-D heap, AVL Tree and RB Tree with 100K operations is 4 ms, 10 ms, 463 ms, and 299 ms when k equals 2, which indicates that Proteus can save 40% running time compared with K-D heap, and significantly better than AVL Tree and RB Tree. For the dequeue operation in Fig. 15(b), we can see that the total running time of Proteus, K-D heap, AVL Tree and RB Tree with 100K operations is 56 ms, 311 ms, 204 ms and 220 ms, when k equals 2, which means that Proteus can save 81% running time compared with K-D heap, and save 74% compared with AVL Tree and RB Tree. In addition, we can observe that the running time of K-D heap, AVL Tree and RB Tree increases significantly when the number of dimensions becomes larger, while that of Proteus with parallel almost keeps the same. The reason is that both AVL Tree and RB Tree are operated in serial, and the running time of each operation increases significantly when the number of dimensions k becomes large. Although K-D heap only keeps one heap as its data structure, the running time to determine the smallest key in the dequeue operation is exponentially proportional to the number of dimensions, which incurs more time overhead especially with a larger k. The running time of Proteus with parallel increases slightly as the computation can be speeded up among the $k-1$ non-primary subheaps.

Fig. 16(a) and Fig. 16(b) show the CDF of running time for the enqueue and dequeue operations when the number of dimensions k equals 2. Here we cannot introduce parallel computation among non-primary subheaps in Proteus. We totally use 100 thousand data points to plot the CDF curve. In Fig. 16(a) and Fig. 16(b), the distribution of Proteus and K-D heap is very close. Both Proteus and K-D heap show faster running time than the AVL and RB tree. The reason is that the additional functions like locating an arbitrary element required by the AVL or RB tree and re-balancing the structures make the factor of $\log n$ larger, leading to the degraded performance.

Finally, we measure the running time for each operation for Proteus, K-D heap, AVL Tree and RB Tree when congestion happens. Table II shows the running time variations for these four data structures, which is evaluated by the CPU clock count via `QueryPerformanceCounter()` API. Each data point is an average of at least ten times. The number of already buffered packets ranges from 1000 to 10000 at the increment of 1000. For enqueue operation, AVL Tree and RB tree take more running time, while for dequeue operation, K-D heap takes more. Both heap and tree data structure are fixed at $\log n$ times. A larger n will increase the times of comparison and swap operations. Table II shows that Proteus performs more stable compared with K-D heap, AVL tree and RB tree. This result also shows that the Proteus can schedule packets quickly even for a large number of buffered packets (*i.e.*, congestion happens). All above results show that heap structure is more efficient for packet scheduling in switches, and sorting all packets in K-D heap incurs more time overhead and is always unnecessary. In a word, Proteus is a perfect choice in the packet scheduling.

VI. CONCLUSION

In this paper, we proposed Proteus, supported by the k-dimension heap structure. It is a low-overhead packets queuing system, which also supports multi-dimensional and arbitrary numbers of ranks. Proteus can be applied to different scheduling strategies and deployed in the software switches. The evaluation results showed that the Proteus can decrease the consumed CPU cycles and storage space.

ACKNOWLEDGMENT

The authors would like to thank anonymous reviewers for their valuable comments. This research is supported by the National Key R&D Program of China 2018YFB1003202, the National Natural Science Foundation of China under Grant Numbers 61772265, 61802172, and 61832005, the Collaborative Innovation Center of Novel Software Technology and

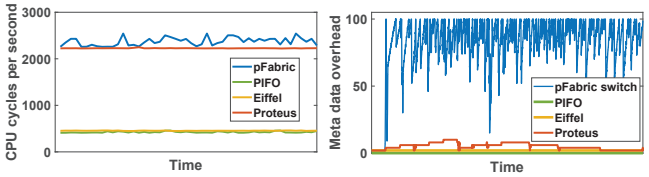
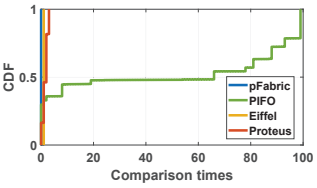
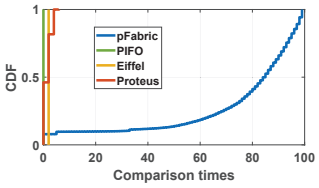


Fig. 12. The CPU cycles consumption comparison.

Fig. 13. The storage overhead comparison.

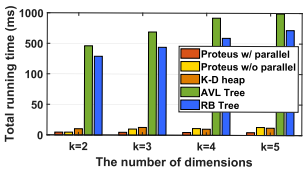


(a) Enqueue operation

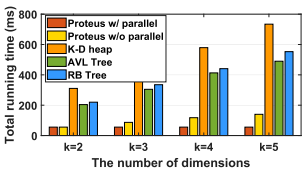


(b) Dequeue operation

Fig. 14. The CDF of comparison times.

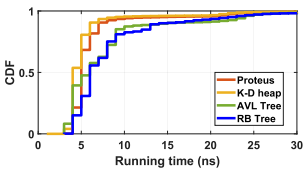


(a) Enqueue operation

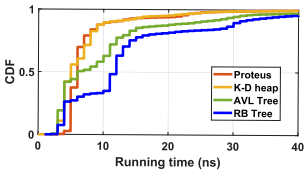


(b) Dequeue operation

Fig. 15. The total running time of 100K operations with different numbers of dimensions.



(a) Enqueue operation



(b) Dequeue operation

Fig. 16. The CDF of performing enqueue and dequeue operations when the number of dimensions is 2.

Industrialization, and the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program.

REFERENCES

[1] B. Montazeri, Y. Li, M. Alizadeh, and J. K. Ousterhout, "Homa: a receiver-driven low-latency transport protocol using network priorities," in *SIGCOMM*, 2018, pp. 221–235.

[2] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian, "DRILL: micro load balancing for low-latency data center networks," in *SIGCOMM*, 2017, pp. 225–238.

[3] B. Vamanan, J. Hasan, and T. N. Vijaykumar, "Deadline-aware datacenter tcp (D2TCP)," in *SIGCOMM*, 2012, pp. 115–126.

[4] P. Veselý, M. Chrobak, L. Jez, and J. Sgall, "A ϕ-competitive algorithm for scheduling packets with deadlines," in *SODA*, 2019, pp. 123–142.

[5] A. Munir, I. A. Qazi, Z. A. Uzmi, A. Mushtaq, S. N. Ismail, M. S. Iqbal, and B. Khan, "Minimizing flow completion times in data centers," in *INFOCOM*, 2013, pp. 2157–2165.

[6] V. Jalaparti, P. Bodík, S. Kandula, I. Menache, M. Rybalkin, and C. Yan, "Speeding up distributed request-response workflows," in *SIGCOMM*, 2013, pp. 219–230.

[7] B. Stephens, A. Akella, and M. M. Swift, "Loom: Flexible and efficient NIC packet scheduling," in *NSDI.*, 2019, pp. 33–46.

[8] B. Stephens, A. Singhvi, A. Akella, and M. M. Swift, "Titan: Fair packet scheduling for commodity multiqueue nics," in *ATC.*, 2017, pp. 431–444.

[9] W. Bai, K. Chen, L. Chen, C. Kim, and H. Wu, "Enabling ECN over generic packet scheduling," in *CoNEXT*, 2016, pp. 191–204.

[10] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal packet scheduling," in *NSDI*, 2016, pp. 501–521.

[11] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: minimal near-optimal datacenter transport," in *SIGCOMM*, 2013, pp. 435–446.

[12] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate," in *SIGCOMM*, 2016, pp. 44–57.

[13] A. Saeed, Y. Zhao, N. Dukkipati, E. W. Zegura, M. H. Ammar, K. Harras, and A. Vahdat, "Eiffel: Efficient and flexible software packet scheduling," in *NSDI*, 2019, pp. 17–32.

[14] "Huawei switch." [Online]. Available: http://support.huawei.com/enterprise/zh/doc/EDOC1000178396?section=j00s

[15] "Catalyst 2960 switch software configuration guide." [Online]. Available: https://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst2960/software/release/12-2_37_se/configuration/guide/scg/swqos.html#wp1028792

[16] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat, "Sincronia: near-optimal network design for coflows," in *SIGCOMM*, 2018, pp. 16–29.

[17] K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S. I. Nikolenko, A. Sirotkin, and P. Eugster, "A programmable buffer management platform," in *ICNP*, 2017, pp. 1–10.

[18] M. T. Jones, *Virtual networking in Linux.*, https://www.ibm.com/developerworks/linux/library/l-virtual-networking/.

[19] "Bess overview." [Online]. Available: https://github.com/NetSys/bess

[20] J. Postel, "Internet protocol," *RFC*, vol. 791, pp. 1–51, 1981.

[21] R. T. Braden, "Requirements for internet hosts - communication layers," *RFC*, vol. 1122, pp. 1–116, 1989.

[22] P. Almquist, "Type of service in the internet protocol suite," *RFC*, vol. 1349, pp. 1–28, 1992.

[23] K. M. Nichols, S. Blake, F. Baker, and D. L. Black, "Definition of the differentiated services field (DS field) in the ipv4 and ipv6 headers," *RFC*, vol. 2474, pp. 1–20, 1998.

[24] B. Tian, C. Tian, H. Dai, and B. Wang, "Scheduling coflows of multi-stage jobs to minimize the total weighted job completion time," in *INFOCOM*. IEEE, 2018, pp. 864–872.

[25] J. Zheng, B. Li, C. Tian, K. Foerster, S. Schmid, G. Chen, and J. Wux, "Scheduling congestion-free updates of multiple flows with chronicle in timed sdns," in *ICDCS*. IEEE Computer Society, 2018, pp. 12–21.

[26] B. Tian, C. Tian, J. Sun, J. Yan, Y. Tang, W. Wang, H. Dai, N. Xia, G. Chen, and W. Dou, "Using the macroflow abstraction to minimize machine slot-time spent on networking in hadoop," in *APNet*, M. Chowdhury and K. Tan, Eds. ACM, 2018, pp. 36–42.

[27] "H3c acl and qos configuration guide." [Online]. Available: http://www.h3c.com/cn/d_201612/963702_30005_0.htm#_Toc468833743

[28] A. Sivaraman, S. Subramanian, A. Agrawal, S. Chole, S. Chuang, T. Edsall, M. Alizadeh, S. Katti, N. McKeown, and H. Balakrishnan, "Towards programmable packet scheduling," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, 2015, pp. 23:1–23:7.

[29] Y. Ding and M. A. Weiss, "The K-D heap: An efficient multi-dimensional priority queue," in *Algorithms and Data Structures, Third Workshop, WADS*, 1993, pp. 302–313.

[30] J. I. Munro and V. Raman, "Sorting multisets and vectors in-place," in *Algorithms and Data Structures, 2nd Workshop WADS*, 1991, pp. 473–480.

[31] R. Sedgewick, *Algorithms*. Addison-Wesley, 1983, ch. Balanced Trees., p. 199.

[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001, ch. Red-black tree., pp. 273–301.

[33] "Programmable packet scheduling at line rate." [Online]. Available: http://web.mit.edu/pifo/#instructions