Pipelined van Emde Boas Tree: Algorithms, Analysis, and Applications

5　30
tianqian@seu.edu.cn

Hao Wang　　Bill Lin

University of California, San Diego, La Jolla, CA 92093–0407. Email: {wanghao,billlin}@ucsd.edu

Abstract—**Priority queues are essential for various network processing applications, including per-flow queueing with Quality-of-Service (QoS) guarantees, management of large fast packet buffers, and management of statistics counters. In this paper, we propose a new data structure for implementing high-performance priority queues based on a pipelined version of the van Emde Boas tree. We show that we can achieve $O(1)$ amortized time operations using our architecture, but we can achieve this algorithmic efficiency using only $O(\log \log u)$ number of pipelined stages, where u is the size of the universe used to represent the priority keys.**

I. INTRODUCTION

A fundamental bottleneck in a number of network processing applications is the real-time maintanence of priority values in sorted-order. Priority queues are usually used for this purpose. Applications include the advanced scheduling of per-flow queues with QoS requirements [1], [2], the management of large fast packet buffers [3], and the management of exact statistics counters [4]. Among these applications, advanced per-flow scheduling with QoS requirements has received the most significant attention. Per-flow queueing is used in advanced high-performance routers to provide isolation between flows by maintaining packets for different flows in separate logical queues. Different flows may have different QoS requirements that correspond to data rates, end-to-end latencies, or cell-loss rate. To provide for QoS guarantees, a number of advanced scheduling techniques have been proposed [1], [2]. Most advanced scheduling techniques are based on assigning timestamps to packets, depending on urgency, and servicing flows in the earliest-timestamp-order. This earliest-timestamp-first scheduling approach can be facilitated using a priority queue that sorts flows in increasing timestamps, where timestamps correspond to priority keys in the priority queue.

A key challenge in the implementation of priority queues is the need to support extremely fast line rates. The speed of networks is increasing at a rapid pace. For example, to support per-flow advanced scheduling on a 10 Gb/s (OC-192) link, with a packet size of 40-bytes, a new entry may be inserted into the priority queue, and an existing entry may be removed once every 32ns. In the advanced QoS scheduling literature, often a binary heap data structure is assumed for the implementation of priority queues [5], which is known to have $O(\log n)$ time complexity for heap operations, where n is the number of heap elements. However, this algorithmic complexity does not scale well with growing queue sizes and is not fast enough for link rates at 10 Gb/s and beyond.

To remedy this performance limitation, pipelined heap data structures have been proposed [6], [7] that can achieve $O(1)$ amortized time complexity. These pipelined heap data structures have a hardware complexity of $O(\log n)$ number of pipelined stages, which can be substantial in hardware complexity for a large n. For example, with $n = 64K$, 16 pipelined stages are required.

In this paper, we propose a new data structure for implementing high-performance priority queues. Our proposed data structure is a pipelined version of a known data structure called a van Emde Boas tree [8]. With the proposed pipelined van Emde Boas tree data structure, we can achieve $O(1)$ amortized time complexity operations. However, the advantage of this new data structure over pipelined heaps is that the hardware complexity is only $O(\log \log u)$ number of pipelined stages, where u is the size of the universe used to represent the priority keys. For example, if 16-bit keys are used to represent timestamps for advanced per-flow scheduling, then the size of the universe would be $u = 2^{16} = 64K$, and $\log \log u$ would only be 4 pipelined stages. This 16-bit precision of priority keys is sufficient to regulate rate of flows on a 10 Gb/s link from a full 10 Gb/s down to the slowest rate of approximately 150 Kb/s.

II. VAN EMDE BOAS TREES

In a binary representation, each data consists of w bits, where $w = \log u$, and u is the size of the universe. This $\log w$ bound of van Emde Boas Trees intuitively suggests a binary search on the bits of a number, and that is essentially what a van Emde Boas Tree does. Suppose x has w bits. Then we split x into two parts: $high(x)$ and $low(x)$, each with $w/2$ bits. For example, let $x = 11001001$, then the split is $high(x) = 1100$ and $low(x) = 1001$.

A. Structure

To handle a universe of size u, we create $\sqrt{u} + 1$ substructures. They are recursively constructed in the same way (they are van Emde Boas structures themselves), as shown in Fig. 1.

1) $\sqrt{u}$ substructures: $(S[0], S[1], \ldots, S[\sqrt{u} - 1])$. Each substructure handles a range of size $\sqrt{u}$ from the universe. A key x is stored in $S[high(x)]$.

2) A single structure $S.summary$ of size $\sqrt{u}$. For every i, if $S[i]$ is nonempty, we store i in the summary structure.

3) $S.min$ is the minimum element of the set. This is not stored recursively. Note that we can test to see if S is empty by simply checking to see if a value is stored in $S.min$.
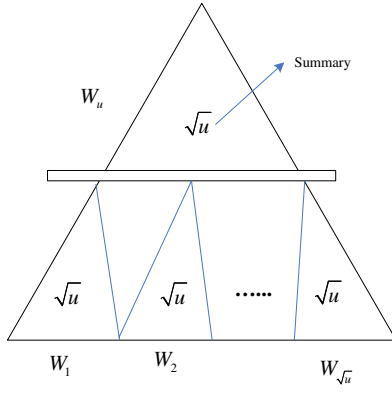
Fig. 1.   An Interpretation of van Emde Boas Trees.

4) $S.max$ is the maximum element of the set. Unlike $S.min$, it is stored recursively.

B. The Widget

The van Emde Boas (vEB) Tree can be characterized as a tree of "widgets". Fig. 2 shows the structure of a widget. The keys to be stored in a van Emde Boas tree are chosen from the universe of keys U.
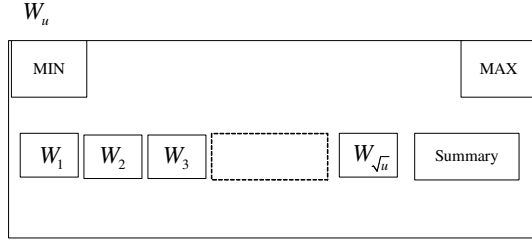

Fig. 2.   van Emde Boas Tree Widget.

1) Each widget is specific to a subrange, over which elements can be searched. The root widget is defined on the entire key range u. We refer to a widget defined on the range u as W_u.

2) Each widget is linked to up to $\sqrt{u}$ sub-widgets, which cover $\sqrt{u}$ sub-ranges of u, respectively. We will refer to them as $W_1, W_2, \ldots, W_{\sqrt{u}}$. There may not be elements in the vEB tree from one or more sub-range in u. Any widget that does not contain any elements is called an empty widget.

3) Each widget contains a maximum (MAX) and minimum (MIN) key in the vEB tree of the sub-range u.

4) Each widget contains a pointer to a summary structure (Summary). The summary structure is also a vEB tree. It is used to answer queries of which sub-widget is the next non-empty sub-widget before/after a given sub-widget W_i.

5) From the recursive structure of the tree, it should be clear that the non-summary leaves of the van Emde Boas tree will be widgets where $MIN = MAX$, and there are no child widgets or summary structure.

C. Interpretation of vEB trees

A van Emde Boas tree can be interpreted in an intuitive manner using complete binary trees, as shown in Fig. 3. Consider a complete binary tree of high h. There are $2^h - 1$ nodes in the tree. Approximately $2^{\lceil h/2 \rceil} - 1$ of these are at depth of $h/2$ or less (i.e. closer to the root than bottom). If there are in total u nodes, the upper subtree (of height $h/2$) is of size approximately $\sqrt{u}$. If we treat the whole tree as a widget, then the summary data structure is the upper subtree and the lower subtrees are its child widgets. Unlike a regular tree, one only need to access the summary or one of the child subtrees to find a successor or predecessor, and hence it follows the sub-logarithmic search time.
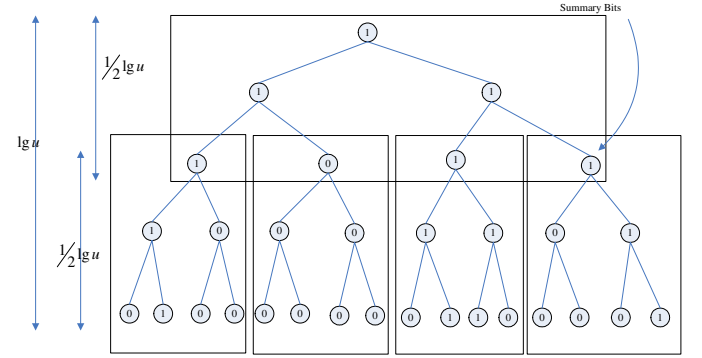

Fig. 3.   Tree view of van Emde Boas (elements are $1, 9, 10, 15$).

III. ALGORITHMS FOR VAN EMDE BOAS TREES

In this section, we describe the operations on a van Emde Boas tree. We then present our pipelined van Emde tree architecture in Section IV along with the algorithms for its operations. In the following pseudo codes, we use W to represent the whole tree structure. All operations described have $O(\log \log u)$ running times.

A. INSERT (x)

The INSERT operation inserts the priority key x into the vEB tree if the key does not already exist.

INSERT(x, W)
1   **if** $x < min[W]$
2   **then** $x \leftarrow min[W]$
3           $min[W] \leftarrow x$
4   **if** MIN$(sub[W][high(x)]) \neq$ NIL
5   **then** INSERT$(low(x), sub[W][high(x)])$
6   **else** $min[sub[W][high(x)]] \leftarrow low(x)$
7   INSERT$(high(x), summary[W])$
8   **if** $x > max[W]$
9   **then** $max[W] \leftarrow x$

In the INSERT operation, each binary data is equally divided into two parts, higher and lower parts. There are two substructures, which correspond to the higher part and lower part. For each loop, it is decided which substructure the new data belongs to. Follow this procedure until each part consists

of only one bit. Even though there are two INSERT calls in its function body, the first call will only be used when the corresponding sub-widget is empty at the time of operation, which happens much less frequently in our consideration.

B. EXTRACTMIN (W)

The EXTRACTMIN operation is a special case of the DELETE operation that deletes and returns the minimum priority key value in the vEB tree. In this part, we use $n[W]$ to represent the number of nonempty entries in the widget W.

EXTRACTMIN(W)
1   **if** $n[W] = 0$
2   **then return** $NULL(m_h, m_l) \leftarrow min(W)$
3   **if** $n(summary[W]) = 1$
4   **then** EXTRACTMIN($summary[W]$)
5        $x_h' \leftarrow min(summary[W])$
6        $x_l' \leftarrow min(sub[W][x_h'])$
7   **else** EXTRACTMIN($sub[W][m_h]$)
8        $x_h' \leftarrow m_h$
9        $x_l' \leftarrow min(sub[W][x_h])$
10 $n[W] = n[W] - 1$
11 $(min)(W) \leftarrow (x_h', x_l')$

Since we always keep track of the minimum value of a van Emde Boas Tree, this operation can be finished in constant time.

C. SUCCESSOR (x, W)

A particularly interesting feature of the vEB tree data structure is that it naturally supports the SUCCESSOR operation, which is very difficult to implement on a binary heap or a pipelined heap. This operation is often useful in network processing applications. Often, we need to find the next entry of operation following the same rule as we used to find the current entry, which can be done using SUCCESSOR operation. The SUCCESSOR operation finds the next smallest priority key larger than x in the vEB tree and returns this key.

SUCCESSOR(x, W)
1   **if** $x < $ MIN(W)
2   **then return** MIN(W)
3   **if** $low(x) < $ MAX($sub[W][high(x)]$)
4   **then** $j \leftarrow$ SUCCESSOR($low(x), sub[W][high(x)]$)
5   **return** $high(x)\sqrt{|W|} + j$
6   **else** $i \leftarrow$ SUCCESSOR($high(x), summary[W]$)
7        $j \leftarrow min[sub[W][i]]$
8   **return** $i\sqrt{|W|} + j$

In the SUCCESSOR operation, we always go back to the upper layer to find the next position that is nonempty. If there is no such entry, the data we access is the last one in the whole vEB Tree structure.

D. EXTRACTSUCC (x, W)

EXTRACTSUCC operation is a special case of the DELETE operation. Instead of deleting and returning the smallest priority key in the vEB tree, the EXTRACTSUCC operation deletes and returns the next smallest priority key

value in the vEB tree greater than x, if such a priority key exists. This is also a very useful operation in network processing applications.

EXTRACTSUCC(x, W)
1   **if** $n[W] = 1$
2   **then return** $NULL$
3   **if** $x < $ MAX(W) and $x > $ MIN(W)
4   **then if** $low(x) < $ MAX($sub[W][high(x)]$)
5   **then** $j \leftarrow$ EXTRACTSUCC
6            $(low(x), sub[W][high(x)])$
7   **return** $high(x)\sqrt{|W|} + j$
8   **else** $i \leftarrow$ EXTRACTSUCC
9            $(high(x), summary[W])$
10        $j \leftarrow min[sub[W][i]]$
11   **return** $i\sqrt{|W|} + j$
12 **else return** MIN(W)
13      $update\ vEB\ Tree$

In the EXTRACTSUCC operation, we find and remove the entry with the smallest value larger than x. If there is no such entry, and vEB Tree is not empty other than the current data we are using, the smallest data of this tree is extracted.

IV. PIPELINED VAN EMDE BOAS TREE

In this section, we present our pipelined van Emde Boas tree and its pipelined operations. The pipeline hardware structure is depicted in Fig. 4.
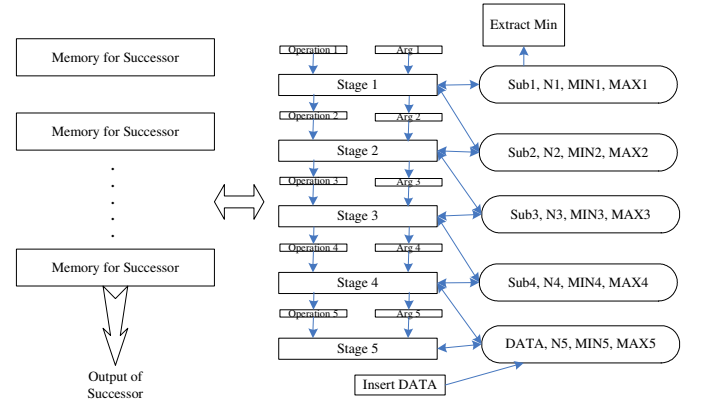

Fig. 4. Simplified block diagram of Pipelined van Emde Boas Trees.

A. Operation Zone

The operation zone is defined as the area of memory in which the operation may access. To get a pipelined algorithm, we have to make sure that no two operations are in the same operation zone at the same time. The key idea is to use MIN and MAX to keep track of the minimum value and maximum value of each widget, and use N to denote the number of elements in each widget. The detailed operations are explained from Section IV-B to Section IV-E.

In the following sections, the pseudo code in the "LOCAL" version of the operation corresponds to a *pipeline stage*, as depicted in Fig. 4. The pseudo code in the "main" procedure of

the operation effectively describes the traversal of the pipeline stages, from the first stage to the last one. In this way, each step of all operations can be divided into different operation zones. After we start the first operation, we may start another one in a later time before the first operation is finished. In this way, a pipelined van Emde Boas Trees is structured.

B. INSERT operation

LOCAL INSERT(x, W)
1   $N[W] + +$
2   **if** $N[W] = 1$
3   **then** $\text{MIN}(W) \leftarrow x$
4   **return** $done$
5   **if** $x < \text{MIN}(W)$
6   **then** $x \leftarrow \text{MIN}(W)$
7        $\text{MIN}(W) \leftarrow x$
8   **if** $x > \text{MAX}(W)$
9   **then** $\text{MAX}(W) \leftarrow x$
10 **if** $\text{MIN}(sub[W][high(x)]) \neq \text{NIL}$
11   **then** $W \leftarrow sub[W][high(x)]$
12      $x \leftarrow low(x)$
13   **return** $unfinished$
14   **else** $\text{MIN}(sub[W][high(x)]) \leftarrow low(x)$
15      $W \leftarrow summary[W]$
16      $x \leftarrow high(x)$
17   **return** $unfinished$

PROCEDURE INSERT(x, W)
1   $global\ x, W$
2   $result \leftarrow unfinished$
3   **while** $result = unfinished$
4   **do** $result \leftarrow$ LOCAL INSERT(x, W)
5   **return** $done$

The key idea with this pipelined version of INSERT operation of van Emde Boas Tree is that we don't have any recursive call. Instead, we use memory to keep track of the position of each operation and its result. Thus, one INSERT operation will be restricted to only local area, i.e. one operation zone, which makes the pipelining idea possible.

C. EXTRACTMIN operation

LOCAL EXTRACTMIN(W)
1   **if** $N[W] = 0$
2   **then return** $done$
3   $N[W] - -$
4   $W' \leftarrow min[summary[W]]$
5   $min[W] \leftarrow min[W']$
6   $W \leftarrow \sqrt{|W|} + min[sub[W][i]]$
7   **return** $unfinished$

EXTRACTMIN(W)
1   $MINIMUM \leftarrow \text{MIN}(W)$
2   $result \leftarrow unfinished$
3   **while** $result = unfinished$
4   **do** $result \leftarrow$ LOCAL EXTRACTMIN(W)

5   **return** $MINIMUM$

In the pipelined EXTRACTMIN operation, there is no recursive call. Memories are used to keep track of the position of the operation. If the tree is empty at the beginning of this operation, the algorithm will terminate immediately.

D. SUCCESSOR operation

LOCAL SUCCESSOR$(x, W, S, flag)$
1   **if** $x < \text{MIN}(W)$
2   **then** $S \leftarrow \text{MIN}(W)$
3   **return** $done$
4   **if** $low(x) < \text{MAX}(sub[W][high(x)])$
5   **then** $W \leftarrow sub[W][high(x)]$
6      $x \leftarrow low(x)$
7      $flag = 1$
8   **else** $W \leftarrow summary[W]$
9      $x \leftarrow high(x)$
10     $flag = 0$
11   **return** $unfinished$

SUCCESSOR$(x, W, S, flag)$
1   $result \leftarrow unfinished$
2   $array\ R, h, choose$
3   $i \leftarrow 0$
4   **while** $result = unfinished$
5   **do** $h[i] \leftarrow high(x)$
6      $R[i] \leftarrow W$
7      $result \leftarrow$ LOCAL SUCCESSOR$(x, W, S, flag)$
8      $choose[i] \leftarrow flag$
9   **while** $i \neq 0$
10   **do if** $choose[i] = 1$
11   **then** $S \leftarrow h[i]\sqrt{|R[i]|} + S$
12   **else** $S \leftarrow S\sqrt{|R[i]|} + S$
13     $i - -$
14 **return** S

There is no recursive call in the pipelined SUCCESSOR operation. Memories are used to keep track of the position of the operation. In order to output the value of the successor we find, we also have to keep track of the path we go through as we are looking for the successor of the current value. The size of this memory will be discussed later.

E. EXTRACTSUCC operation

LOCAL EXTRACTSUCC$(x, W, S, flag)$
1   **if** $x < \text{MIN}(W)$ or $x > \text{MAX}(W)$
2   **then** $S \leftarrow \text{MIN}(W)$
3      $remove\ Min(W)$
4      $update\ vEB\ Tree$
5   **return** $done$
6   **if** $low(x) < \text{MAX}(sub[W][high(x)])$
7   **then** $W \leftarrow sub[W][high(x)]$
8      $x \leftarrow low(x)$
9      $flag = 1$
10   **else** $W \leftarrow summary[W]$

```
11            x ← high(x)
12            flag = 0
13            return unfinished
```

EXTRACTSUCC($x, W, S, flag$)

```
 1   result ← unfinished
 2   array R, h, choose
 3   i ← 0
 4   while result = unfinished
 5     do h[i] ← high(x)
 6        R[i] ← W
 7        result ← LOCAL EXTRACTSUCC(x, W, S, flag)
 8        choose[i] ← flag
 9   while i ≠ 0
10     do if choose[i] = 1
11          then S ← h[i]√|R[i]| + S
12          else  S ← S√|R[i]| + S
13        i − −
14   return S
```

It is also true here that there is no recursive call in the pipelined EXTRACTSUCC operation. Memories are used to keep track of the memory positions of the operation. If there is no such successor entry, and vEB Tree is not empty other than the current data we are using, the smallest data of this tree is extracted.

V. MEMORY MANAGEMENT OF PIPELINED VEB TREE

A. Memory

On the right hand side of Fig. 4 are the memories for each stage. We have to use two time slots to finish one stage. That is because EXTRACTMIN operation may need to access the memory of the next stage. When an entry of a widget is extracted, an entry from the corresponding sub-widget will be need to replace it. Given two time slots, we can guarantee that no two consecutive EXTRACTMIN operations will access the same time slot at the same time. The structure of the each memory is shown in Fig. 5.
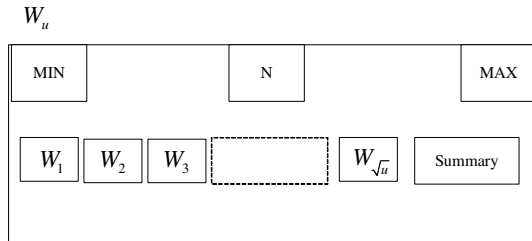

Fig. 5.   Pipelined van Emde Boas Tree Widget in Memories.

In all the memories except the last one, MIN, MAX and N are stored. Also, the links of summary to each W_i are kept. It is worth noting that there is no data in these memories. All the data is kept in the memory of the last processor. Data is modified only in the last step. The size of the memory in stage 1 is $O(u^{1/2})$. It is $O(u^{3/4})$ for stage 2, $O(u^{7/8})$ for stage 3,

and $O(u^{15/16})$ for stage 4. The final stage in the example in Fig. 4, which is stage 5, is of size $O(u)$.

Consider a universe of size 2^{20}. The stages are of size 2^{10}, 2^{15}, $2^{17.5}$, $2^{18.75}$, and 2^{20}. In the last stage, we store all the data. So the memory for the last stage is much larger than all the others. We can put them all in SRAM to achieve better performance.

On the left hand side of Fig. 4 are the memories for the operation of successor. There are $(\log\log u)$ operations allowed in this structure at the same time, and $\log\log u$ stages. So the size of the memories matrix is of the order of $O(\log\log u^2)$. Actually, since finding successor operation is not required in some applications, we can achieve a even simpler pipelined structure without this memory.

B. Operations

The pipelined van Emde Boas Tree works like this. First, we input operation 1. After one time slot, all the parameters will be transfered to the next processor. To avoid the possible collision in EXTRACTMIN operation, no new operation will be input in stage 1. After one more time slot, when operation 1 moves to stage 3, a new operation 2 can be input into stage 1. In this way, we can make sure that there will be no collision in this pipelined structure.

The number of processors required in this structure is $\log\log u$. Given a universe of 1 Gigabyte, we will need to have 10 processors working in parallel. The amount of operations supported in this structure is $\log\log u/2$. Thus, constant time operation is realized.

A better method is to divide each operation into three clock cycles. They are: a) reading from memory; b) comparing two values of next stage memory to find the minimum; c) writing this minimum into the memory of current stage. Therefore, operations can go down the stages at a rate of one or more operations every three cycles. It is not one operation every two cycles, because it is possible to have consecutive EXTRACTMIN operations. However, we need the memory to support a reading throughput of two entries per cycle, and an additional writing throughput of one entry per cycle.

REFERENCES

[1] A. K. Parekh, R. G. Gallager, "A generalized processor sharing approach to flow control in integrated service networks: The single-node case," *IEEE/ACM Transactions on Networking*, vol. 1, pp. 334-357, 1993.

[2] A. Demers, S. Keshav, S. Shenkar, "Analysis and simulation of a fair queueing algorithms," *Proceedings of SIGCOMM'89*, pp. 1-12, Austin, TX, Sept. 1989.

[3] S. Iyer, N. Mckeown, "Designing buffers for router line cards," *Stanford University HPNG Technical Report - TR02-HPNG-031001*, Stanford, CA, Mar. 2002.

[4] D. Shah, "Analysis of a statistics counter architecture," *Proc. IEEE Hot Interconnects 9*, IEEE CS Press, Los Alamitos, CA, 2001.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to algorithms," McGraw-Hill Book Company, ISBN 0-07-013143-0.

[6] R. Bhagwan, B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," *Proceedings of INFOCOM 2000*, Tel Aviv, Israel, 2000.

[7] A. Ioannou, M. Katevenis, "Pipelined heap (priority queue) management for advanced scheduling in high-speed networks," *Proceedings of IEEE Int. Conf. on Communications (ICC 2001)*, 2001.

[8] P. van Emde Boas, "Design and implementation of an efficient priority queue," *Math. Syst. Theory*, 10:99-127, 1977.