Succinct Priority Indexing Structures for the Management of Large Priority Queues

Hao Wang and Bill Lin
University of California San Diego, La Jolla, CA 92093

***Abstract*—Priority queues are an essential building block for implementing advanced per-flow service disciplines at high-speed network links. In this paper, we propose novel solutions to the scalable implementation of priority queues by decomposing the problem into two parts, a succinct priority index in SRAM that can efficiently maintain a real-time sorting of priorities, coupled with a DRAM-based implementation of large packet buffers. In particular, we propose three related novel succinct priority index data structures for implementing high-speed priority indexes: a Priority-Index (PI), a Counting-Priority-Index (CPI), and a Pipelined Counting-Priority-Index (Pipelined CPI). We show that all three structures can be very compactly implemented in SRAM using only $\Theta(U)$ space, where U is the size of the universe required to implement the priority keys (timestamps). We also show that our proposed priority index structures can be implemented very efficiently as well by leveraging hardware-optimized instructions that are readily available in modern 64-bit microprocessors. The operations on the PI and CPI structures take $\Theta(\log_W U)$ time, where W is the processor word-length (i.e., $W = 64$ bits). Alternatively, operations on the Pipelined CPI structure take constant time with only $\Theta(\log_W U)$ pipeline stages. Finally, we show the application of our proposed priority index structures for scalable management of large packet buffers at line speeds.**

I. Introduction

A fundamental bottleneck in a number of network processing applications is the real-time maintenance of priority values in sorted order. Priority queues are usually used for this purpose. Especially in the advanced scheduling of per-flow queues with Quality-of-Service (QoS) requirements, priority queues have received the most attention. To provide for QoS guarantees, a number of advanced scheduling techniques have been proposed [1], [2].

Scalable priority queue implementation requires solutions to two fundamental problems. The first is the ability to sort queue elements in real-time at ever increasing line speeds. In the advanced QoS scheduling literature, often a binary heap data structure is assumed for the implementation of priority queues [4], which is known to have $\Theta(\log_2 n)$ 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 40 Gb/s and beyond. To remedy the performance limitation, pipelined heap data structures have been proposed [5], [6], [7] that can achieve constant amortized time complexity with $\Theta(\log_2 n)$ pipelined stages, which can be prohibitively expensive in hardware complexity for a large n. For example, with $n = 16$ million, at least 24 pipelined stages are required. The Pipelined van Emde Boas Tree structure [8], [9] has also been proposed, which is based on the van Emde Boas tree [9]. It achieves constant amortized time complexity with $\Theta(\log_2 \log_2 U)$ pipelined stages, where U is the size of the universe representing the range of the priority keys. With a universe of size $U = 16$ million, only 5 pipelined stages are required. However, in comparison to our proposed data structures in this paper, the pipelined van Emde Boas tree requires considerably more complicated operations and more storage memory.

The second problem of the scalable priority queue implementation is the ability to store a huge number of packets. The problem motivated the work in [11] where they proposed a heap-based hybrid SRAM/DRAM priority queue system. In particular, they adopted a previously developed hybrid SRAM/DRAM FIFO memory architecture [12] in their proposed solution. However, this solution is hard to implement and requires a large amount of SRAMs.

In this paper, we propose novel solutions to solve the scalable priority queue implementation problem by decomposing it into two parts, a succinct priority index in SRAM that can efficiently maintain the real-time sorting of priorities, coupled with memory management of large packet buffers. In particular, we propose three related novel succinct priority index data structures for implementing high-speed priority indexes: a Priority-Index (PI), a Counting-Priority-Index (CPI), and a Pipelined Counting-Priority-Index (Pipelined CPI). We show that all three structures can be compactly implemented in SRAM using only $\Theta(U)$ space, where U is the size of the universe required to implement the priority keys (deadline timestamps). For example, for $U = 16$ million, which is sufficient to regulate rate of flows on a 40 Gb/s link from the full rate down to the slowest rate of approximately just 2.5 Kb/s, only about 2 MB of SRAM suffices to implement these priority index structures. We also show that our proposed priority index structures can be implemented very efficiently as well by leveraging hardware-optimized instructions that are readily available in modern 64-bit microprocessors. In particular, operations on the PI and CPI structures can be realized in $\Theta(\log_W U)$ time, where W is the word-length of the processor (i.e., $W = 64$ bits). Alternatively, operations on the Pipelined CPI structure can be realized in constant time using only $\Theta(\log_W U)$ pipeline stages (e.g., only 4 pipeline stages for $U = 16$ million). Although the $\Theta(\log_W U)$ hardware complexity is not as good asymptotically in comparison to the pipelined van Emde boas tree structure, the hardware complexity of our Pipelined CPI design is actually lower for any universe $U \leq 2^{30}$ when $W = 64$, which is practically the case for all relevant applications. Finally, we show that

978-1-4244-3876-1/09/$25.00 ©2009 IEEE

our proposed priority index structures can be combined with a DRAM-based architecture to provide scalable storage for large packet buffers at line speeds.

The remainder of this paper is organized as follows. In Section II, we introduce the basic data structure called a Priority-Index (PI). Although simple, this basic data structure is not amenable to pipelining. In Section III, we present an extended data structure called a Counting-Priority-Index (CPI) that can be readily pipelined, and the pipelined version of this data structure, called a Pipelined Counting-Priority-Index (Pipelined CPI), is described in Section IV. In Section V, we discuss the memory management issues using DRAM to implement large packet buffers. Finally, we conclude in Section VI.

II. Succinct Priority Index Abstraction

In this section, we outline a high-level abstraction that we call a succinct priority index, which is a bitmap data structure that can conceptually be interpreted as a tree. Although the priority indexing structure is presented in this paper in the context of advanced per-flow scheduling, we note that they can also be applied to other networking problems, such as statistics counting [3], which also relies on priority queues as a fundamental building block.

A. Structure

The basic structure of a succinct priority index abstraction, which we call a Priority-Index (PI), is a perfect W-way tree as shown in Fig. 1. Suppose we have a fixed universe of size $U = W^h$, and we wish to represent a set S of N elements from this universe, where $N \leq U$. The basic abstraction can be viewed as a bitmap that is used to record which elements of the universe are present in S. Each element i of the universe is associated with a binary bit b_i. Bit b_i is set to 1 if $i \in S$, and 0 otherwise. In a PI structure, a leaf node in the perfect W-way tree contains W bits of b_i, for $0 \leq i \leq U$. For a non-leaf node, there are also W bits. Each of these W bits is associated to a corresponding child node. The bits of the non-leaf nodes are used as a *summary* of their child nodes – i.e., a bit in a non-leaf node is set to 1 if there is at least one non-zero bit in its child node, or 0 otherwise. As an example, a PI of 3 levels in a universe $U = 2^{18}$ for 18-bit priority indexes with $W = 64$ is shown in Fig. 1.

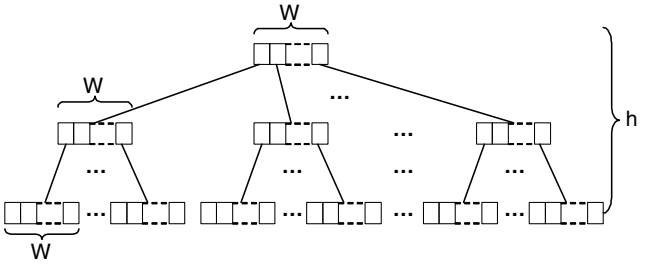

Fig. 1. Data Structure of PI with $h = 3$

B. Supported Operations

The succinct priority indexing abstraction supports the operations shown in Table I. Modern 64-bit x86 processors (from both Intel and AMD [13], [14]) all have built-in instructions to compute the position of the *most-significant-bit* set using `BSR` (bit-scan-reverse) or *least-significant-bit* set using `BSF` (bit-scan-forward) as a single step operation. If there is no such bit, the `BSR` and `BSF` operations will set flag `ZF` (zero flag) to 1. In addition, AMD also has a `LZCNT` (leading-zero-count) instruction [14] that returns the number of leading 0's. Then the position of the most significant bit in a word is just `LZCNT` $+ 1$, unless `LZCNT` is 64, in which case all bits are 0's and there is no value 1 bit in the word. Therefore, the position of the most-significant-bit or least-significant-bit can be located in constant time.

TABLE I
Operations supported by succinct priority index architecture

`test(i)`	Test if index i is in set S
`insert(i)`	Insert a new index i to set S
`delete(i)`	Delete index i from set S
`findmin`	Find the smallest index in set S
`findmax`	Find the largest index in set S
`extractmin`	Delete the smallest index in set S
`extractmax`	Delete the largest index in set S
`successor(i)`	Find the successor of index i in set S
`predecessor(i)`	Find the predecessor of index i in set S
`extractsucc(i)`	Delete the successor of index i in set S
`extractpred(i)`	Delete the predecessor of index i in set S

In the succinct priority index architecture, index values increase from left to right and decrease from right to left, as opposed to the heap structure where index values increase from top to bottom (or bottom to top). In this paper, only `findmin`, `extractmin`, `successor` and `extractsucc` operations are described in details. The `findmax`, `extractmax`, `predecessor` and `extractpred` operations can be derived using right-left symmetry.

III. Counting-Priority-Index

In this section, a new data structure Counting-Priority-Index (CPI) is presented. In a CPI, all the operations of a succinct priority index are supported in a top-down fashion.

A. Structure

In CPI, at each non-leaf level node, a total of W counters are needed. Let the counters for a given non-leaf node be `counter`$[0]$, `counter`$[1]$, $\cdots$, `counter`$[W-1]$. The `counter`$[i]$ of a node is used to track the number of value 1 bits in its $(i+1)^{th}$ child node from the left. For a counter, if there is no value 1 bit in its child nodes, it should be 0, and also the corresponding bit in the W-bit word is reset to 0. Otherwise the counter should be non-zero and the corresponding bit in the W-bit word is set to 1. The data structure of CPI with height $h = 3$ is shown in Fig. 2.

B. Operations

The `test` operation is implemented in constant time on the leaf-level nodes. The `insert`, `findmin`, and `findmax` operations function in a top-down fashion. For the `insert(i)` operation, as the operation moves down from the root to the

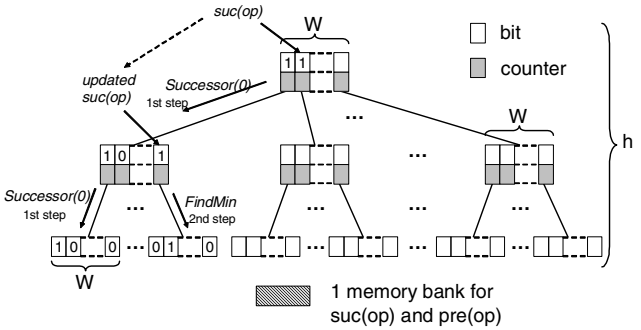

Fig. 2. Example of `successor(0)` operation in CPI

leaf-level nodes, the counters on its path are incremented by one. If a counter of a non-leaf node is zero before the `insert` operation, the corresponding bit in the W-bit word of the node is set to 1 from 0. The `delete(i)` operation in CPI starts from root node and goes down the tree until it reaches the index i. At each non-leaf level node, the counters on its path is decreased by one. If the counter reaches zero, the corresponding bit in the W-bit word of the current node is reset to 0. Otherwise, the W-bit word is left unchanged. At the leaf-level node, the bit corresponding to index i is set to be 0 and the index number i is used as the output. The `extractmin` operation can be treated as the combination of `findmin` and `delete` operation.

The `successor(i)` operation works in a greedy way consisting of two steps. In the first step, it starts from root node and works top-down to find the path to index i. At the same time, an extra variable `suc(op)` is used to remember the possible bit that can lead to the successor of index i. Parameter `op` is the *operation id* number in the system that is unique for each operation and it is initialized to `NULL`. If there are other value 1 bits in the same node to the right of the path leading to index i, the `suc(op)` is updated to remember the location of the first one of such bits from the left. Otherwise, the `suc(op)` is left unchanged. After index i is reached at the leaf-level, the operation continues in the second step by first checking if the successor of i is in the same leaf node as i. If it is true, then the operation is finished and the successor of index i is found. Otherwise, the operation resumes from the location kept in `suc(op)` to find the minimum index in S using `suc(op)` as root node, and then outputs the index as the successor of index i. Therefore the second step is the same as `findmin` using `suc(op)` as the root node. An example of the `successor(i)` operation in CPI with $i = 0$ is shown in Fig. 2. The `extractsucc(i)` operation can be implemented similarly.

C. Memory and Complexity

Assume a CPI of size $U = W^h$ with $W = 64$. There are U bits at the leaf level. For the upper next level, there are U/W bits for the W-bit words. And there are U/W counters each of size $\log_2 W$, so together $6U/W$ bits for the counters. In the same way, the next upper level needs $7U/W^2$ bits, and so on. Moreover, extra memory bank is needed to keep variable `suc(op)` and `pre(op)`. Only $\log_2 h$ bits are used to record the level of each variable and $\log_2 W$ bits are used to keep the location of each variable within a node. Thus the total memory size N is,

$$N = \{U + \frac{7U}{W} + \cdots + \frac{7U}{W^{h-1}}\} + \{\log_2 h + \log_2 W\} \quad (1)$$

$$\approx 1.11U \quad (2)$$

There is only a size of $1.11U$ memory required to keep the priority indexing of the CPI for a universe of size U. In a CPI, the time complexity for all of the operations is $\Theta(\log_W U)$.

IV. Pipelined Counting-Priority-Index

In this section, the Pipelined Counting-Priority-Index (Pipelined CPI) is presented. In a Pipelined CPI, all supported operations are also implemented in a top-down fashion and can be pipelined. In each time slot, a new operation can be initiated in a Pipelined CPI.

A. Structure

By pipelining all operations, a pipelined CPI can achieve higher access rate than a regular CPI. The data structure of a Pipelined CPI with $h = 3$ is shown in Fig. 3.

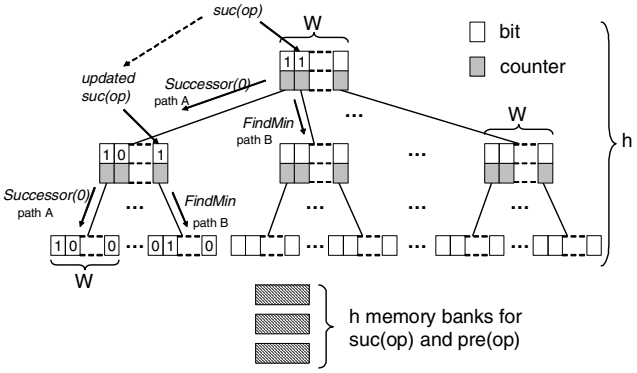

Fig. 3. Example of `successor(0)` operation in Pipelined CPI

B. Operations

The `test(i)` operation can finish in constant time. `insert(i)`, `delete(i)`, `findmin(i)`, `findmax(i)`, `extractmin(i)`, and `extramax(i)` operations are top-down as in CPI. In order to support pipelined operations, `successor(i)`, `predecessor(i)`, `extractsucc(i)` and `extractpred(i)` operations need to be modified.

The `successor(i)` operation is modified based on the one in CPI to make the best use of the potential pipeline speed. Once the `suc(op)` is not `NULL` at a node with W-bit word at a certain level, the `successor(i)` operation is separated into two parallel sub-operations following two different paths. One path (path `A`) is the path of another `successor(i)` operation which uses the current bit as its root node. The other path (path `B`) is the path of the `findmin` operation which uses the `suc(op)` as its root node. If the `suc(op)` is updated in the path `A`. Path `B` stops its current process and continues the `findmin` operation using the new `suc(op)` as its root node. As a consequence, these two sub-operations are always working at the same level from the root of the Pipelined CPI until

they reach the leaf-level nodes. At the leaf-level node, if the suc(op) is still NULL, the operation outputs SNF (successor not found). If the suc(op) is updated at leaf-level, it uses the new suc(op) index as the output of successor(i), because this means the successor of index i and index i itself are in the same leaf-level node. Otherwise it uses the output of path B as the output of the successor(i) operation. The flow graph of the successor(i) operation is shown in Fig. 4.

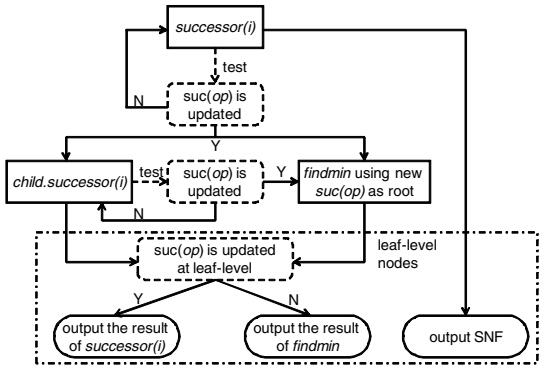

Fig. 4. Flow graph of successor(i) operation in Pipelined CPI

An example of the pipelined successor(i) operation with $i = 0$ is shown in Fig. 3. Here the suc(op) is updated twice, once at the root level, and the second time at the second level. After each update, the findmin operation resumes from the new suc(op).

The extractsucc(i) operation cannot be easily pipelined. During the operation, the counters on the path B need to be updated. Since part or all of the path B may not be the path that leads to the successor of index i, some updated counters may need to be recovered. Therefore it is necessary to separate one extractsucc(i) operation into one succeccsor(i) operation with returned index j and delete(j) operation. In this way, the counters will not be updated unless it is on the path leading to the index j.

C. Memory and Complexity

Compared with non-pipelined CPI, in Pipelined CPI, extra memory banks are needed to keep track of more variables suc(op) and pre(op). As in CPI, $\log_2 h$ bits are used to keep the level of each variable and $\log_2 W$ bits are used to keep the location of each variable within a node. There are at most h operations in the Pipelined CPI at the same time, so h such variables are enough for the pipeline operations. For $W = 64$, the following holds for the total memory size N.

$$N = \{U + \frac{7U}{W} + \cdots + 7W\} + h\{\log_2 h + \log_2 W\} \quad (3)$$

$$\approx 1.11U + h\log_2 h + \log_2 W^h \quad (4)$$

$$\approx 1.11U + \log_2 U, \text{ since } h \ll W \quad (5)$$

$$\approx 1.11U \text{ if } U > 2^{10}. \quad (6)$$

Using Pipelined CPI, $\Theta(1)$ time operations can be achieved. At each level of the Pipelined CPI, two sub-operations may happen at the same time on different nodes for successor and predecessor operations. Therefore if all the nodes in the same level of the tree are in a single memory bank, this memory bank needs to support *double* access rate per time slot. On the other hand, if different nodes in the same level of the tree are in different memory banks, each memory bank only needs to support *single* access rate per time slot. When the Pipeline CPI is employed to achieve a constant processing speed, each level should be stored in an independent SRAM unless speedup is used in the SRAM.

V. Large DRAM-Based Priority Queues

In this section, we describe the application of our proposed succinct priority index data structures to the management of large DRAM-based packet buffers in which the priorities correspond to *unique* departure times. To store a large number packets, DRAMs are necessary to provide affordable bulk storage. However, worst-case access times of DRAM devices are too slow to match the line rates of high-performance routers. Suppose $1/b$ is the ratio between the worst-case access bandwidth of DRAMs and the line rate, then we can match the line rate requirement by operating b DRAM banks in parallel. Although the worst-case access time for each DRAM bank is still b cycles, a new memory operation can be initiated in every cycle if it is initiated to a different DRAM bank in an interleaving manner. The idea of using the interleaving of DRAM banks has been explored in [15] for the management of large packet buffers with per-flow queueing. We adapt this idea for storing packets that will be serviced in earliest-deadline-first order.

In particular, we uniquely locate a packet in memory based on its unique departure time. However, we cannot simply *stripe* packet locations across the memory banks in timestamp order for two reasons (i.e., map a packet with departure time t to the k^{th} DRAM bank where $k = t \mod b$). First, the departure timestamps of arriving packets may not follow striping order since the corresponding packets may have different service requirements. Therefore, there can be pathological cases in which consecutively arriving packets may be written to the same DRAM bank. Second, a packet may depart *earlier* than its departure timestamp when a link is idle. Suppose the current time is t, but the packet with the earliest departure deadline has a departure time of $j > t$. Then in this case, we can service this packet in the current cycle by *advancing virtually* the clock to j. Here again, the read access pattern from the DRAM banks may not follow striping order.

The basic idea in dealing with these issues is to randomly distribute the timestamp locations evenly across the b memory banks so that with high probability that each memory bank will receive about one out of b write (read) operations to it on average. This is achieved by applying a pseudorandom permutation function $\pi : \{1, \ldots, N\} \rightarrow \{1, \ldots, N\}$ to a packet timestamp to obtain a permuted memory location. We then use a simple location policy with a packet where the departure time t will be stored in the k^{th} DRAM bank, where $k = \pi(t) \mod b$, at address location $a = \lfloor \pi(t)/b \rfloor$. This is depicted in Fig. 5.

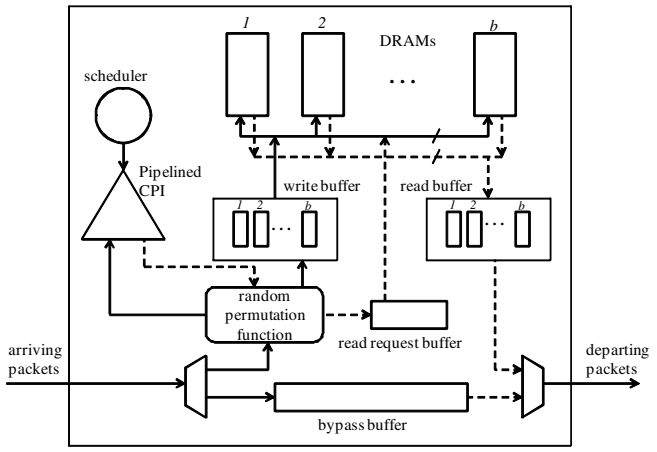

Fig. 5. Memory Management of Large Priority Queues

Specifically, when a new packet arrives with a departure timestamp of i, a write operation is generated to the k^{th} DRAM bank, $k = \pi(i) \mod b$, to location $a = \lfloor \pi(i)/b \rfloor$. We also set the corresponding i^{th} bit in a priority index (using for example a Pipelined CPI). Then, at every time slot, we lookup the priority index to find the packet with the next earliest deadline j to service. This operation can be efficiently realized thanks to the hardware-optimized $\texttt{extractsucc(t)}$ operation supported by our priority index structures, which finds the next bit set at or after the bit location corresponding to the current time t. We can then retrieve the corresponding packet from k^{th} DRAM bank, $k = \pi(j) \mod b$, at location $a = \lfloor \pi(j)/b \rfloor$. Accordingly, we clear the corresponding j^{th} bit in the priority index.

For the write operations, we maintain a write request/packet buffer for the incoming packets waiting to be stored. This buffer is composed of b queues, one queue for each DRAM bank. The write requests at the heads of the queues are serviced in an interleaving order. Both the requests and the actual packets are stored in the queues. To bound the size of the buffer, we assume that the arrival process for the incoming write requests going to a certain DRAM bank (after random permutation) follows the behavior of an M/D/1 system. This is because Internet traffic can be thought of as the superposition of independent traffic sources modeled as point processes [16]. Correspondingly, the probability that a queue in the SRAM write buffer will overflow some threshold x can be analyzed using the steady state probability of the unfinished work exceeding a certain level x for a Poisson source as a surrogate. This overflow probability can be derived as the overflow probability of a corresponding M/D/1 system. For example, to ensure an overflow probability of 10^{-9} for incoming packet traffic load up to 90%, it is sufficient for each queue in the write buffer to hold $L = 100$ packets. The read operation is implemented similarly.

We employ a *departure buffer* in SRAM of size $K > T$ entries to hold packets retrieved from the DRAM until their actual departure times. This departure buffer is implemented as a circular buffer in which the buffer location is mapped from the departure time slot. We actually size the departure buffer to have $K = 2 \times T = 2 \times L \times b$ entries to account for the situation where the departure time of a packet is less than $K = 2 \times T = 2 \times L \times b$ cycles away from its arrival time. In this case, there might not be sufficient time to write the packet to the DRAM and retrieve it back since the worst-case total round-trip time is $K = 2 \times T = 2 \times L \times b$ cycles. To address this issue, we simply write the packet directly to the SRAM departure buffer at the corresponding departure time slot location so that the packet is immediately available for retrieval. Effectively, the departure buffer acts as a *bypass buffer* in such situations.

VI. Conclusion

In this paper, fast and scalable succinct data structures for the implementation of priority queues in networking applications are presented. The presented data structures and the associated algorithms are well-suited for modern 64-bit x86 processors, which have hardware-optimized instructions that can be leveraged. These structures can be very compactly implemented in SRAM using only $\Theta(U)$ space, where U is the size of the universe required to implement the priority keys. Our Pipelined CPI data structure can effectively support constant time priority management operations. In addition to being very fast, the architecture also scales well to a large number of priority values and to large queue sizes. The hardware complexity is only $\Theta(\log_W U)$, where Moreover, the Pipelined CPI can be combined with an interleaved DRAM-based architecture to provide scalable storage for huge priority queues at line speeds.

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 Trans. Netw.*, vol. 1, pp. 334-357, 1993.

[2] A. Demers, S. Keshav, S. Shenkar, "Analysis and simulation of a fair queueing algorithms," *in Proc. ACM SIGCOMM 1989*, Austin, TX, Sep. 1989.

[3] D. Shah, "Analysis of a statistics counter architecture," *IEEE Symp. on Hot Interconnects* , Los Alamitos, CA, Aug. 2001.

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

[5] R. Bhagwan, B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," *in Proc. IEEE INFOCOM 2000*, Tel Aviv, Israel, Mar. 2000. 2000.

[6] A. Ioannou, M. Katevenis, "Pipelined heap (priority queue) management for advanced scheduling in high-speed networks," *IEEE/ACM Trans. Netw.*, vol. 15, pp. 450-461, 2007.

[7] H. Wang, B. Lin, "On the Efficient Implementation of Pipelined Heaps for Network Processing," *in Proc. IEEE GLOBECOM 2006*, San Francisco, CA, Nov., 2006.

[8] H. Wang, B. Lin, "Pipelined van Emde Boas tree: Algorithms, analysis, and applications," *in Proc. IEEE INFOCOM 2007*, Anchorage, AK, May 2007.

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

[10] C. Villamizar, C. Song, "High performance tcp in ansnet," *ACM Computer Communication Review*, vol. 24, no. 5, pp. 45-60, 1994.

[11] X. Zhuang, S. Pande, "A scalable priority queue architecture for high speed network processing," *In Proc. IEEE INFOCOM 2006*, Barcelona, Spain, Apr. 2006.

[12] S. Iyer, N. Mckeown, "Analysis of a memory architecture for fast packet buffers," *In Proc. IEEE HPSR 2001*, Dallas, TX, May 2001.

[13] Intel 64 and IA-32 architectures software developer's manual, volume 2B, Nov. 2007.

[14] Software optimization guide for AMD family 10h processors, Apr. 2008.

[15] G. Shrimali, N. McKeown, "Building Packet Buffers using Interleaved Memories," *In Proc. IEEE HPSR 2005*, Hong Kong, China, May 2005.

[16] J. Cao, K. Ramanan, "A Poisson limit for buffer overflow probabilities," *In Proc. IEEE INFOCOM 2002*, New York, NY, Jun. 2002.