Recursive Design of Hardware Priority Queues[☆]

Y. Afek[a,*], A. Bremler-Barr[b], L. Schiff[a,1]

[a] *Tel Aviv University, Tel Aviv, Israel*
[b] *The Interdisciplinary Center, Hertzelia, Israel*

Abstract

A recursive and fast construction of an n-element priority queue from exponentially smaller hardware priority queues and size n RAM is presented. All priority queue implementations to date require either $O(\log n)$ instructions per operation or, exponential (with key size) space or, expensive special hardware whose cost and latency dramatically increases with the priority queue size. Hence constructing a priority queue (PQ) from considerably smaller hardware priority queues (which are also much faster) while maintaining the $O(1)$ steps per PQ operation is critical. Here we present such an acceleration technique called the Power Priority Queue (PPQ) technique. Specifically, an n-element PPQ is constructed from $2k - 1$ primitive priority queues of size $\sqrt[k]{n}$ ($k = 2, 3, ...$) and a RAM of size n, where the throughput of the construct beats that of a single, size n primitive hardware priority queue. For example an n-element PQ can be constructed from either three $\sqrt{n}$ or five $\sqrt[3]{n}$ primitive H/W priority queues.

Applying our technique to a TCAM based priority queue, results in TCAM-PPQ, a scalable perfect line rate fair queuing of millions of concurrent connections at speeds of 100 Gbps. This demonstrates the benefits of our scheme; when used with hardware TCAM. We expect similar results with systolic arrays, shift-registers and similar technologies.

As a byproduct of our technique we present an $O(n)$ time sorting algorithm in a system equipped with a $O(w\sqrt{n})$ entries TCAM, where here n is the number of items, and w is the maximum number of bits required to represent an item, improving on a previous result that used an $\Omega(n)$ entries TCAM. Finally, we provide a lower bound on the time complexity of sorting n-element with TCAM of size $O(n)$ that matches our TCAM based sorting algorithm.

1. Introduction

A priority queue (PQ) is a data structure in which each element has a priority and a dequeue operation removes and returns the highest priority element in the queue. PQs are the most basic component for scheduling, mostly used in routers, event driven simulators and is also useful in shortest path and navigation (e.g. Dijkstra's algorithm) and compression (Huffman coding). In

[☆] A preliminary version of this paper appeared in the proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures, (SPAA 2013) [1].

[*] Corresponding author

Email addresses: `afek@post.tau.ac.il` (Y. Afek), `bremler@idc.ac.il` (A. Bremler-Barr), `schiffli@post.tau.ac.il` (L. Schiff)

[1] Supported by European Research Council (ERC) Starting Grant no. 259085.

time based scheduling systems, time values, such as customer arrival time or, expected end of service time, are transformed into priorities that are then used in the PQ [2, 3].

As noted first by Kleinrock, packet scheduling schemes are at the foundations of the successful construction of computer networks [4, 2, 5]. In today's routers and switches, PQs play a critical role in scheduling and deciding the transmission order of packets [6, 7, 8]. Priority Queues are used to enforce fairness while also considering the different priorities of flows, thus guaranteeing that flows get a weighted (by their relative importance and history of usage) fair share of the bandwidth independent of the size of packets used.

Since PQs share the same time bounds as sorting algorithms[9], in high throughput scenarios, (e.g., backbone routers) special hardware PQs are used. Hardware PQs are usually implemented by ASIC chips that are specially tailored and optimized to the scenario and do not scale well [10, 11, 12, 13, 14, 15].

We present a new construction for large hardware PQs, called Power Priority Queue (PPQ), which recursively uses small hardware priority queues in parallel as building blocks to construct a much larger one. The size of the resulting PQ is a power of the smaller PQs size, specifically we show that an n elements priority queue can be constructed from only $2k - 1$ copies of any base (hardware) $\sqrt[k]{n}$ elements (size) priority queue. Our construction benefits from the optimized performance of small hardware PQs and extends these benefits to high performance, large size PQ.

We demonstrate the applicability of our construction in the case of the Ternary Content Addressable Memory (TCAM) based PQ, that was implied by Panigrahy and Sharma[16]. The TCAM based PQ, as we investigate and optimize in Appendix E , has poor scalability and becomes impractical when it is required to hold 1M items. But by applying our construction with relatively tiny TCAM based PQ, we achieve a PQ of size 1M with throughput of more than 100M operations per second, which can be used to schedule packets at a line rate of 100Gb/s. The construction uses in parallel 10 TCAMs (or TCAM blocks) of size 110Kb and each PQ operation requires 3.5 sequential TCAM accesses (3 for Dequeue and 4 for Insert).

Finally this work also improves the space and time performance of the TCAM based sorting scheme presented in [16]. As we show in Section 4 an n elements sorting algorithm is constructed from two $w\sqrt{n}$ entries TCAM's, where w is the number of bits required to represent one element (in [16] two n entries TCAM's are used). The time complexity to sort n elements in our solution is the same as in [16], $O(n)$, when counting TCAM accesses, however our algorithm accesses much smaller TCAM's and thus is expected to be faster. Moreover, in Section 4.2 we prove a lower bound on the time complexity of sorting n elements with a TCAM of size n (or $\sqrt{n}$) that matches our TCAM based sorting algorithm.

2. Priority Queues Background

2.1. Priority queues and routing

Since the beginning of computer networks, designing packet scheduling schemes has been one of the main difficulties [5, 2]. In today's routers and switches, PQs play a critical role in scheduling and deciding the order by which packets are forwarded [6, 7, 8]. Priority Queues is the main tool with which the schedulers implement and enforce fairness combined with priority among the different flows. Guaranteeing that flows get a weighted (by their relative importance) fair share of the bandwidth independent of packet sizes they use.

For example, in the popular Weighted Fair Queueing (WFQ) scheduler, each flow is given a different queue, ensuring that one flow does not overrun another. Then, different weights are associated with the different flows indicating their levels of quality of service and bandwidth allocation. These weights are then used by the WFQ scheduler to assign a time-stamp to each arriving packet indicating its virtual finish time according to emulated Generalized Processor Sharing (GPS). And now comes the critical and challenging task of the priority queue, to transmit the packets in the order of the lowest timestamp packet first, i.e., according to their assigned timestamps[2]. For example, in a 100Gbps line rate, hundreds of thousands of concurrent flows are expected[3]. Thus the priority queue is required to concurrently hold more than million items and to support more than 100 million insert or dequeue operations per second. Note that the range of the timestamps depends on the router's buffer size and the accuracy of the scheduling system. For best accuracy, the timestamps should at least represent any offset in the router's buffer. Buffer size is usually set proportional to $RTT \cdot lineRate$, and for a 100Gbps line rate and RTT of 250ms, timestamp size can get as high as 35 bits.

No satisfactory software PQ implementation exists due to the inherent $O(\log n)$ step complexity per operation in linear space solutions, or alternatively $O(w)$ complexity but then with $O(2^w)$ space requirement, where n is the number of keys (packets) in the queue and w is the size of the keys (i.e., timestamps in the example above). These implementations are mostly based on binary heaps or Van De Boas Trees[12]. None of these solutions is scalable, nor can it handle large priority queues with reasonable performances.

Networking equipment designers have therefore turned to two alternatives in the construction of efficient high rate and high volume PQ's, either to implement approximate solutions, or to build complex hardware priority queues. The approximation approach has light implementation and does not require a PQ [18]. However the inaccuracy of the scheduler hampers its fairness, and is thus not applicable in many scenarios. The hardware approaches, described in detail in the next subsection, are on the other hand not scalable.

2.2. Hardware priority queue implementations

Here we briefly review three hardware PQ implementations, Pipelined heaps [13, 19], Systolic Arrays [10, 11] and Shift Registers [15]. ASIC implementations, based on pipelined heaps, can reach $O(1)$ amortized time per operation and $O(2^w)$ space [13, 19], using pipeline depth that depends on w, the key size, or $\log n$ the number of elements. Due to the strong dependence on hardware design and key size, most of the ASIC implementations use small key size, and are not scalable for high rate. In [20] a more efficient pipelined heap construction is presented, and our technique resembles some of the principals used in their work, however their result is a complex hardware implementation requiring many hardware processors or special elements and is very specific to pipelined heaps and of particular size, while the technique presented here is general, scalable with future technologies and works also with simpler hardware such as the TCAM.

Other hardware implementations are Systolic Arrays and Shift Registers . They are both based on an array of $O(n)$ comparators and storing units, where low priority items are gradually pushed to the back and highest priority are kept in front allowing to extract the highest priority item in $O(1)$ step complexity. In shift register based implementations new inputs are broadcasted to all units where as in systolic arrays the effect of an operation (an inserted item, or values shift) propagates from the front to the back one step in each cycle. Shift Registers require a global communication board that connects with all units while systolic arrays require bigger units to

[2]Note that it's enough to store the timestamp of the first packet per flow.

[3]Estimated by extrapolating the results in [17] to the current common rate.

hold and process propagated operations. Since both of them requires $O(n)$ special hardware such as comparators, making them cost effective or even feasible only for low n values and therefore again not scalable.

Another forth approach, which is mostly theoretical is that of Parallel Priority Queues. It consists of a pipeline or tree of processors [21], each merges the ordered list of items produced by its predecessor processor(s). The number of processors required is either $O(n)$ in a simple pipeline or $O(\log n)$ in a tree of processors, where n is the maximal number of items in the queue. The implementations of these algorithms [22] is either expensive in case of multi-core based architectures or unscalable in the case of ASIC boards.

3. PPQ - The Power Approach

The first and starting point idea in our Power Priority Queue (PPQ) construction is that to sort n elements one can partition them into $\sqrt{n}$ lists of size $\sqrt{n}$ each, sort each list, and merge the lists into one sorted list. Since a sorted list and a PQ are essentially the same, we use one $\sqrt{n}$ elements PQ to sort each of the sublists (one at a time), and a second $\sqrt{n}$ elements PQ in order to merge the sublists. Any $\sqrt{n}$ elements (hardware) PQ may be used for that. In describing the construction we call each PQ that serves as a building block, Base Priority Queue (BPQ). This naive construction needs two $\sqrt{n}$ elements BPQ's to construct an n element PPQ.

The BPQ building block expected API is as follows:

- Insert(item) - inserts an item with priority item.key.
- Delete(item) - removes an item from the BPQ, item may include a pointer inside the queue.
- Dequeue() - removes and returns the item with the highest priority (minimum key).
- Min() - like a peek, returns the BPQ item with the minimum key.

Note that the Min operation can easily be constructed by caching the highest priority item after every Insert and Dequeue operation, introducing an overhead of a small and fixed number of RAM accesses.

In addition our construction uses a simple in memory (RAM) FIFO queue, called RList, implemented by a linked list that supports the following operations:

- Push(item) - inserts an item at the tail of the RList.
- Pop() - removes and returns the item at the head of the RList.

Notice that an RList FIFO queue, due to its sequential data access, can be mostly kept in DRAM while supporting SDRAM like access speed (more than 100Gb/s). This is achieved by using SRAM based buffers for the head and tail parts of each list, and storing internal items in several interleaved DRAM banks [23].

3.1. Power Priority Queue

To construct a PPQ (see Figures 1 and 2) we use one BPQ object, called input-BPQ, as an input sorter. It accepts new items as they are inserted into the PPQ and builds $\sqrt{n}$ long lists out of them. When a new $\sqrt{n}$ list is complete it is copied to the merging area and the input BPQ starts constructing a new list. A second BPQ object, called exit-BPQ, is used to merge and find the minimum item among the lists in the merge area. The pseudo-code is given in Appendix A. The minimum element from each list in the merge area is kept in the exit-BPQ. When the

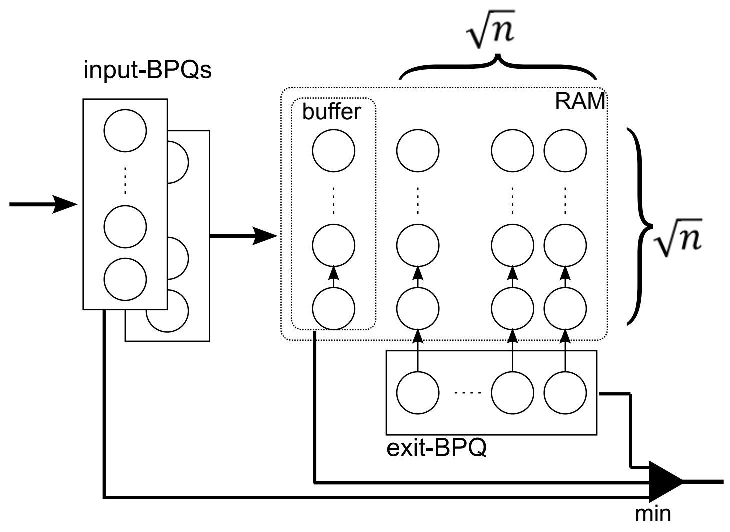


Figure 1: The basic (and high level) Power Priority Queue (PPQ) construction. Note that the length of sublists in the RAM may reach $2\sqrt{n}$ (after merging).

minimum element in the exit-BPQ is dequeued as part of a PPQ dequeue, a new element from the corresponding list in the merging area is inserted into the exit-BPQ object. Except for the minimum of each RList sorted list the elements in the merging area are kept in a RAM (see notice at the end of the previous subsection). Each PPQ Dequeue operation extracts the minimum element from the exit-BPQ (line 37) or the input-BPQ (line 46), depending on which one contains the smallest key.

The above description suffers from two inherent problems (bugs); first, the construction may end up with more than $\sqrt{n}$ small RLists in the merging area which in turn would require an exit-BPQ of size larger than $\sqrt{n}$, and second, how to move $\sqrt{n}$ sorted elements from a full input-BPQ to an RList while maintaining an $O(1)$ worst case time per operation. In the next subsections we explain how to overcome these difficulties (the pseudo-code of the full algorithm is given in Appendix A).

3.1.1. Ensuring at most $\sqrt{n}$ RLists in the RAM

As items are dequeued from the PPQ, RAM lists become shorter, but the number of RAM lists might not decrease and we could end up with more than $\sqrt{n}$ RLists, many of which with less than $\sqrt{n}$ items. This would cause the exit-BPQ to become full, even though the total number of items in the PPQ is less than n. To overcome this, any time a new list is ready (when the input-BPQ is full) we find another RAM list of size at most $\sqrt{n}$ (which already has a representative in the exit-BPQ) and we start a process of merging these two lists into one RList in the RAM (line 22 in the pseudo-code) keeping their mutual minimum in the exit-BPQ (lines 25-28), see Figure 2(c). In case their mutual minimum is not the currently stored item in the exit-BPQ, the stored item should be replaced using exit-BPQ.Delete operation, followed by an Insert of the mutual minimum.

This RAM merging process is run in the background interleaving with the usual operation of the PPQ. In every PPQ.Insert or PPQ.Dequeue operation we make two steps in this merging (line 13), extending the resulting merged list (called *fused-sublist* in the code) by two more items. Considering the fact that it takes at least $\sqrt{n}$ insertions to create a new RAM sublist, we are guaranteed that at least $2\sqrt{n}$ merge steps complete between two consecutive RAM lists creations, ensuring that the two RAM lists are merged before a new list is ready. Note that since the heads of two merged lists and the tail of the resulting list are buffered in SRAM the two merging steps have small, if any at all, influence on the overall completion time of the operation.

If no RAM list smaller than $\sqrt{n}$ exists then either there is free space for the new RAM list and there is no need for a merge, or the exit-BPQis full, managing $\sqrt{n}$ RAM lists of size larger than $\sqrt{n}$, i.e., the PPQ is overfull. If however such a smaller than $\sqrt{n}$ RLists exists we can find one such list in $O(1)$ time by holding a length counter for each RList, and managing an *unordered* set of small RLists (those with length at most $\sqrt{n}$). This set can easily be managed as a linked list with $O(1)$ steps per operation.

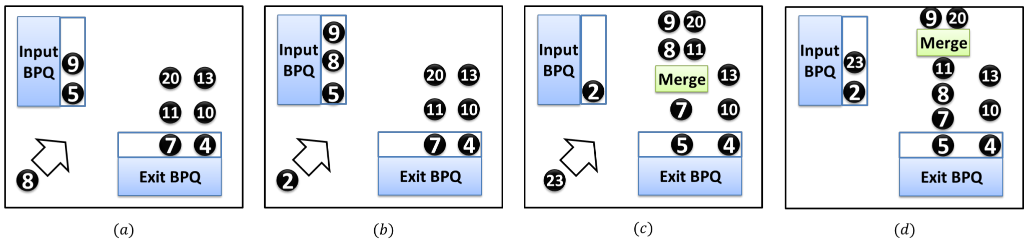


Figure 2: A sequence of operations, Insert(8), Insert(2), and Insert(23), and the Power Priority Queue (PPQ) state after each ((*b*)-(*d*)). Here $n = 9$ and the Merge in state (c) is performed since there is a sublist whose size is at most $\sqrt{n}$.

3.1.2. Moving a full input-BPQ into an RList in the RAM in $O(1)$ steps

When the input-BPQ is full we need to access the $\sqrt{n}$ sorted items in it and move them into the RAM (either move or merge with another RList as explained above). At the same time we also need to use the input-BPQ to sort new incoming items. Since the PPQ is designed for real time scheduling systems, we should carry out these operations while maintaining $O(1)$ worst case steps per insert or dequeue operations. As the BPQ implementation might not support an operation “copy all items and reset” in one step, the items should be deleted (using dequeue) and copied to the RAM one by one. Such an operation consumes too much time ($\sqrt{n}$) to be allowed during a single Insert operation. Therefore, our solution is to use two input-BPQs with flipping roles, while we insert a new item to the first we evacuate one from the second into an RList in the RAM. Since their size is the same, by the time we fill the first we have emptied the second and we can switch between them. Thus our construction uses a total of three BPQ objects, rather than two. Note that when removing the highest-priority element, we have to consider the minimums of the queues and the list we fill, i.e., one input-BPQ, one RList and the exit-BPQ.

The pseudo-code of the full algorithm is provided in Appendix A. The two input-BPQs are called input-BPQ[0] and input-BPQ[1], where input-BPQ[*in*] is the one currently used for insertion of new incoming items and input-BPQ[*out*] is evacuated in the background into an RList named buffer[*out*]. The RList accessed by buffer[*in*] is the one being merged with another small sublist already in the exit-BPQ.

3.2. PPQ Complexity Analysis

Here we show that each PPQ.Insert operation requires at most 3 accesses to BPQ objects, which can be performed in parallel, thus adding one sequential access time, and each PPQ.dequeue operation requires at most 2 sequential accesses to BPQ objects.

The most expensive PPQ operation is an insert in which exactly the input-BPQ[*in*] becomes full. In such an operation the following 4 accesses (A1-A4) may be required; A1: An insert on input-BPQ[*in*], A2: a Delete and A3: Insert in the exit-BPQ, and A4: A dequeue from the input-BPQ[*out*]. Accesses A2& A3 are in the case that the head item in the new list that starts its merge with an RList needs to replace an item in the exit-BPQ. However, notice that accesses A1, A2 and A4 may be executed in parallel, and only access A3 sequentially follows access A2. Thus the total sequential time of this PPQ.Insert is 2. Since such a costly PPQ.Insert happens only once every $\sqrt{n}$ Insert operations, we show in Appendix B how to delay access A3 to a subsequent PPQ.Insert thus reducing the worst case sequential access time of PPQ.Insert to 1.

The PPQ.Dequeue operation performs in the worst case a Dequeue followed by an Insert to the exit-BPQ and in the background merging process, a Dequeue in one input-BPQ. Therefore the PPQ Dequeue operation requires in the worst case 3 accesses to the BPQ objects which can be performed in two sequential steps.

Both operations can be performed with no more than 7 RAM accesses per operation (which can be made to the SRAM whose size can be about 8MB), and by using parallel RAM accesses, can be completed within 6 sequential RAM accesses. Thus, since each packet is being inserted and dequeued from the PPQ the total number of sequential BPQ accesses per packet is 3 with 6 sequential SRAM accesses. This can be farther improved by considering that the BPQ accesses of the PPQ.Insert are to a different base hardware object than those of the PPQ.Dequeue. In a balanced Insert-Dequeue access pattern, when both are performed concurrently, this reduces to 2 the number of sequential accesses to BPQ objects per packet.

3.3. The TCAM based Power Priority Queue (TCAM-PPQ)

The powering technique can be applied to several different hardware PQ implementations, such as, Pipelined heaps [13, 19], Systolic Arrays [10, 11] and Shift Registers [15]. Here we use a TCAM based PQ building block, called TCAM Ranges based PQ (RPQ), to construct a TCAM based Power Priority Queue called TCAM-PPQ, see Figure 3. The RPQ construction is described in Appendix D, it is an extension of the TCAM based set of ranges data structure of Panigrhay and Sharma[16] and features a constant number of TCAM accesses per RPQ operation using two $w \cdot m$ entries TCAMs (each entry of w bits) to handle m elements. Thus a straightforward naive construction of an n items TCAM-PPQ requires 6 TCAM's of size $w\sqrt{n}$ entries.

Let us examine this implementation in more detail. According to the RPQ construction in Appendix D (also summarized in Table D.2) 1 sequential access to TCAMs is required in the implementation of RPQ.Insert, 1 in the implementation of RPQ.Dequeue and 3 for RPQ.delete(item). Combining these costs with the analysis in the previous subsection yields that the worst case cost of TCAM-PPQ.Insert is 3 sequential accesses to TCAMs, and also 3 for TCAM-PPQ. Dequeue. However, TCAM-PPQ.Insert costs 3 only once every $\sqrt{n}$ inserts, i.e., its amortized cost converges to 2, and the average of the two operations together is thus 2.5 sequential TCAM accesses. Note that it is possible to handle priorities' (values of the PQ) wrap around by a simple technique as described in Appendix E.3.

Consider for example the following use case of the TCAM-PPQ. It can handle a million keys in a range of size 2^{35} (reasonable 100 Gbps rate [24]) using 6 TCAMs, each smaller than 1 Mb.

Considering a TCAM rate of 500 millions accesses per second (reasonable rate for 1 Mb TCAM [25]), and 2.5 accesses per operation (Insert or Dequeue) this TCAM-PPQ works at a rate of 100 million packets per second. Assuming average packet size of 140 bytes [13, 26], then the TCAM-PPQ supports a line rate of 112 Gbps.

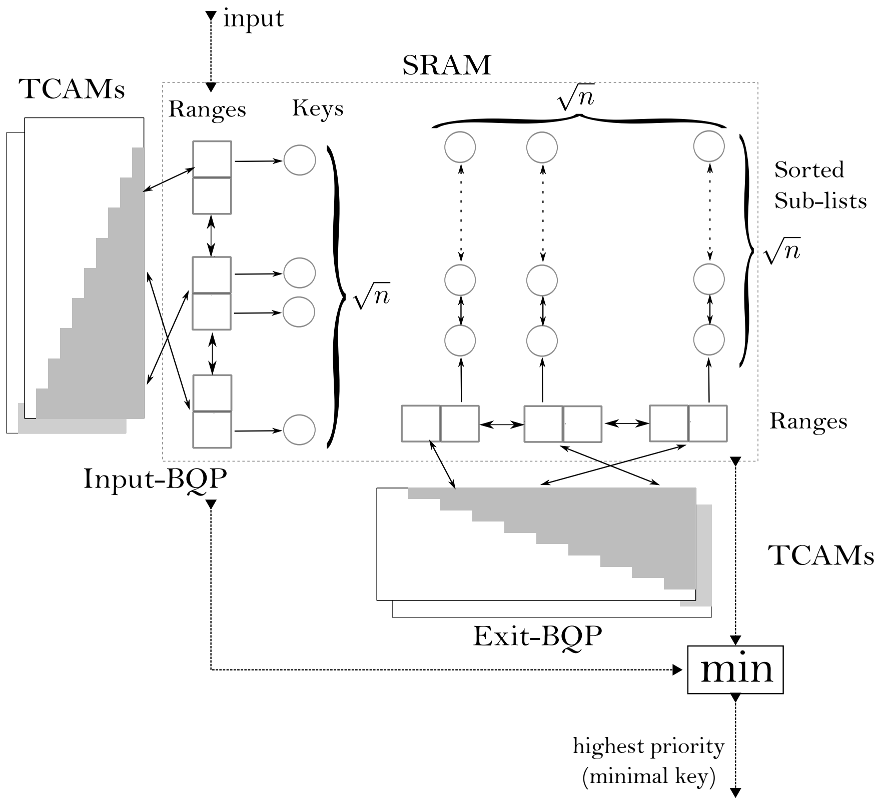
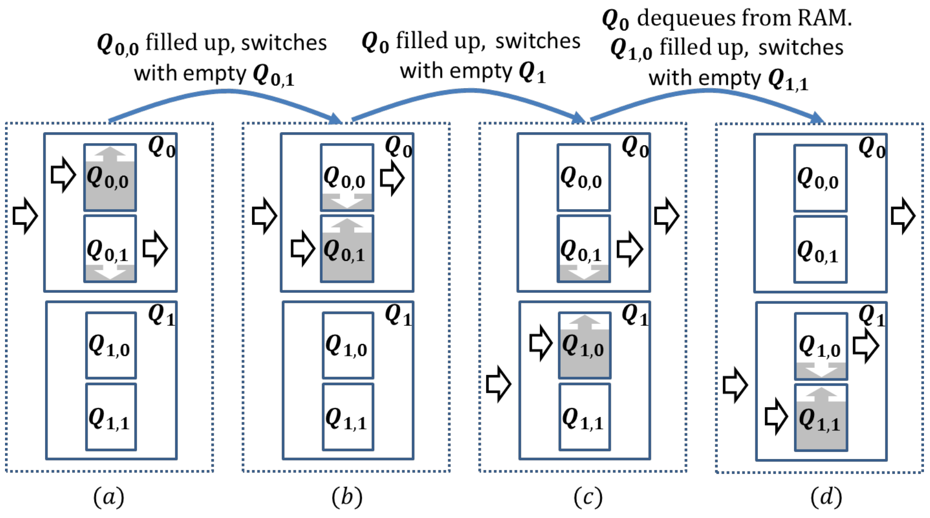


Figure 3: The TCAM based Priority Queue (TCAM-PPQ) construction.

3.4. The Power k Priority Queue - PPQ(k)

The PPQ scheme describes how to build an n-element priority queue from three $\sqrt{n}$ elements priority queues. Naturally this calls for a recursive construction where the building blocks are built from smaller building blocks. Here we implement this idea in the following way; (see Figure 5) we fix the size of the exit-BPQ to be x, the size of the smallest building block. In the RAM area x lists each of size n/x are maintained. The input-BPQ is however constructed recursively. In general if the recursion is applied k times, a PPQ with capacity n is constructed from $\cdot 2^k$ BPQs each of size $\sqrt[k]{n}$.

However, a closer look at the BPQ's used in each step of the recursion reveals that each step requires only 2 size x exit-BPQ and each pair of input-BPQs is replaced by a pair of input-BPQs whose size is x times smaller as illustrated in Figure 4. Thus each step of the recursion adds only 2 size x BPQ's objects (the exit-BPQs) and the corresponding RAM space (see Figure 4).



$Q_{0,0}$ filled up, switches with empty $Q_{0,1}$
Q_0 filled up, switches with empty Q_1
Q_0 dequeues from RAM. $Q_{1,0}$ filled up, switches with empty $Q_{1,1}$
Q_0
$Q_{0,0}$
$Q_{0,1}$
Q_1
$Q_{1,0}$
$Q_{1,1}$
(a)
(b)
(c)
(d)

Figure 4: A scenario in which four n/x^2 input-BPQs construct two size n/x input-BPQs that in turn are used in the construction of one size n input-BPQ. As explained in the text it illustrates that only 2 n/x^2 input-BPQs are required at any point of time.

At the last level 2 size x input-BPQs are still required. Consider the top level of the recursion as illustrated in Figure 4, where a size n PPQ is constructed from two input-BPQs, Q_0 and Q_1 each of size n/x and each with size n/x RAM (the RAM and the exit-BPQs are not shown in the figure at any level). Each of Q_0 and Q_1 is in turn constructed from two size n/x^2 input-BPQs ($Q_{0,0}$, $Q_{0,1}$, $Q_{1,0}$, and $Q_{1,1}$) and the corresponding RAM area and size x exit-BPQ. As can be seen, at any point of time only two n/x^2 input-BPQs are in use. For example moving from state (b) to state (c) in Figure 4, $Q_{0,0}$ is already empty when we just switch from inputting into Q_0 to inputting to Q_1, and Q_1 needs only $Q_{1,0}$ for n/x^2 steps. When Q_1 starts using $Q_{1,1}$, moving from (c) to (d), $Q_{0,1}$ is already empty, etc. Recursively, these two size n/x^2 input-BPQs may thus be constructed by two n/x^3 input-BPQs. Moreover notice that since only two input-BPQs are used at each level, also only two exit-BPQs are required at each level. The construction recurses k times until the size of the input-BPQ equals x, which can be achieved by selecting $x = \sqrt[k]{n}$. Thus the whole construction requires $2k - 1$, size $\sqrt[k]{n}$ BPQs. In our construction in Section 3.4.1 we found that $k = 3$ gives the best performance for a TCAM based PQ with 100GHz line rate.

We represent the time complexity of an operation $OP \in \{ins, deq\}$ on a size n PPQ(k) built from base BPQs of size $x = \sqrt[k]{n}$, $T(OP, n, x)$, by a three dimensional vector $(N_{ins}, N_{deq}, N_{del})$ that represents the number of BPQ Insert, the number of BPQ Dequeue and the number of BPQ Delete operations (respectively) required to complete OP in the worst case. BPQ operations, for for moderate size BPQ, are expected to dominate other CPU and RAM operations involved in the algorithm. In what follows we show that the amortized cost of an Insert operation is (1,1,1/x) (i.e., all together at most 3 sequential BPQ operations), and (1,1,0) for a Dequeue operation.

If we omit the Background routine, each PPQ(k) Dequeue operation either performs a Dequeue from input-BPQ[in] (a PPQ ($k - 1$) of size n/x), extract an item from the exit-BPQ (using one BPQ Dequeue and one Insert operations) or fetch it from a buffer[out] (no BPQ operation).

Therefore we can express the time complexity of PPQ(k) Dequeue operations (without Background), $t(deq, n, x)$ or in shorter form $t_{deq}(n)$, by the following recursive function:

$$t_{deq}(n) = \begin{cases} (0,0,0) & \text{min. is in buffer}[out] \\ (1,1,0) & \text{min. is in exit-BPQ} \\ t_{deq}(n/x) & \text{otherwise} \end{cases} . \tag{1}$$

Considering the fact that a priority queue of capacity x is the BPQ itself, $t_{deq}(x) = t(deq, x, x) = (0, 1, 0)$. Therefore the worst case time for any Dequeue is $(1, 1, 0)$, i.e. $t(deq, n, x) = (1, 1, 0)$ when $n > x$.

Note that the equation $t(deq, n, x) = (1, 1, 0)$ expresses the fact that Dequeue essentially updates at most one BPQ (holding the minimum item), which neglects the RAM and CPU operations required to find that BPQ within the $O(k)$ possible BPQs and buffers. Neglecting these operations is reasonable when k is small, or when we use additional BPQ-like data structure of size $O(k)$ that holds the minimums of all input-BPQ[in] and buffers and can provide their global minimum in $O(1)$ time.

The Background() routine, called at the end of the Dequeue operation, recursively performs a Dequeue from all input-BPQ[out]s. Since there are $k - 1$ input-BPQ[out]s, the Background()'s time cost, $B(n, x)$, equals $(k-1, k-1, 0)$. Therefore the total time complexity of PPQ(k) Dequeue (by definition $T(deq, n, x) = t(deq, n, x) + B(n, x)$) equals k BPQ Dequeues and k BPQ Inserts in the worst case, i.e.

$$T(deq, n, x) = (k, k, 0). \tag{2}$$

If we omit the Background routine, each PPQ(k) Insert operation performs an Insert to one of its two n/x-sub-queues (the input-BPQ[in]) and sometimes (when the input-BPQ[in] is full) also starting merging of a new RList with existing one which might require a Delete and Insert to the exit-BPQ. Therefore we can express the time complexity of PPQ(k) Insert operation (without Background), $t(ins, n, x)$ or in shorter form $t_{ins}(n)$, by the following recursive function:

$$t_{ins}(n) = \begin{cases} t_{ins}(n/x) + (1,0,1) & \text{input-BPQ}[in] \text{ is full} \\ t_{ins}(n/x) & \text{otherwise} \end{cases} . \tag{3}$$

Considering the fact that a priority queue of capacity x is the BPQ itself, $t_{ins}(x) = t(ins, x, x) = (1, 0, 0)$. Therefore the worst case time of any Insert is $(k, 0, k - 1)$, i.e. $t(ins, n, x) = (k, 0, k - 1)$ when $n > x$. When we include the cost of the Background, we get that

$$T(ins, n, x) = (2k - 1, k - 1, k - 1). \tag{4}$$

Moreover, since the probability that at least one input-BPQ[in] is full is approximately $1/x$, the amortized cost of a PPQ(k) Insert without Background is $(1, 0, 0) + \frac{1}{x}(1, 0, 1)$, and with background it is $(k, k - 1, 0) + \frac{1}{x}(1, 0, 1)$.

An important property of the Background() routine is that it only accesses input-BPQ[out]s while the rest of the operations of Insert and Dequeue access input-BPQ[in]s, therefore it can be executed in parallel with them. Moreover, since Background performs a Dequeue on input-BPQ[out]s, and since in input-BPQ[out] minimum key can be found locally (no input-BPQ[in] is used by input-BPQ[out]), all Dequeue calls belonging to a Background can be performed concurrently, thereby achieving parallel time cost of (1,1,0) for the Background routine. As a consequence, putting it all together, in a fully parallel implementation the amortized cost of Insert is $(1, 1, 1/x)$ and $(1, 1, 0)$ for Dequeue.

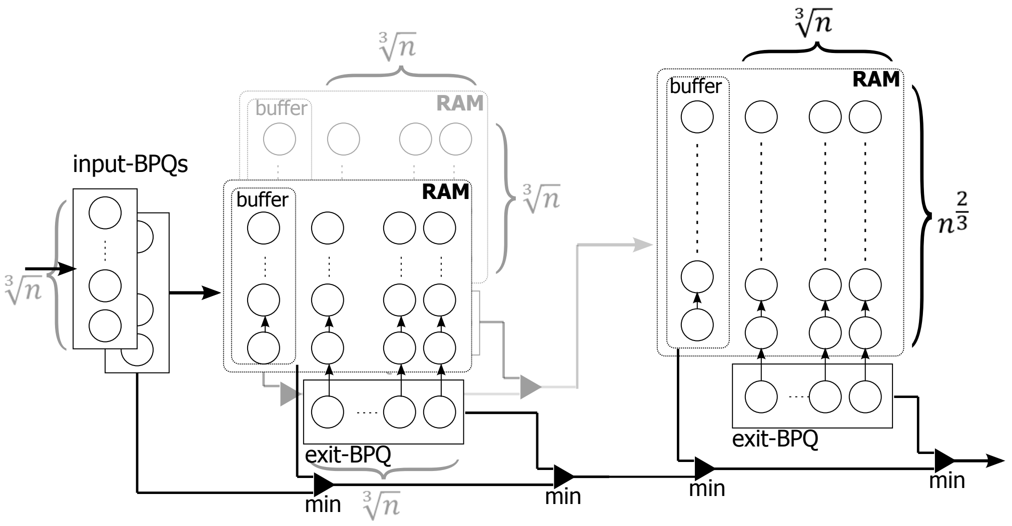


Figure 5: High level diagram of the Power $k = 3$ Priority Queue - PPQ(3) construction.

3.4.1. The generalized TCAM-PPQ(k)

When applying the PPQ(k) scheme with the RPQ (Ranges based Priority Queue), we achieve a priority queue with capacity n which uses $O(wk\sqrt[k]{n})$ entries TCAM (each entry of size w bits) and $O(k)$ TCAM accesses per operation. More precisely, using the general analysis of PPQ(k) above and the RPQ analysis summarized in D.2, TCAM-PPQ(k) requires $2k - 1$ RPQs of size $\sqrt[k]{n}$ each and achieves Insert with amortized cost of $3k - 1$ TCAM accesses and Dequeue with $3k$ TCAM accesses. As noted above these results can be farther improved, by using parallel execution of independent RPQ operations, which when fully applied can results in this case with only 3 TCAM accesses.

Since access time, cost and power consumption of TCAMs decreases as the TCAM gets smaller, the TCAM-PPQ(k) scheme can be used to achieve an optimized result based on the goals of the circuit designer. Note that large TCAMs also suffer from long sequential operation latency which leads to pipeline based TCAM usage. The reduction of TCAM size with TCAM-PPQ(k) allows a simpler and straightforward TCAM usage. Considering the TCAM size to performance tradeoffs the best TCAM based PQ is the TCAM-PPQ (3) whose performance exceeds RPQ and simple TCAM based lists implementations.

Let $T(S)$ be the access time of a size S TCAM, then another interesting observation is that for any number of items n, the time complexity of each operation on TCAM-PPQ(k) is $O\left(k \cdot T(\theta\sqrt[k]{n})\right)$, where θ is either w or w^2 depending on whether the TCAM returns the longest prefix match or not, respectively. This time complexity can be also expressed by $O\left(\log n \cdot \frac{T(S)}{\log S - \log\theta}\right)$. This implies that faster scheduling can be achieved by using TCAMs with lower $T(S)$ to $(\log S - \log\theta)$ ratio, suggesting a design objective for future TCAMs.

The new TCAM-PPQ(3) can handle a million keys in a range of size 2^{35} (reasonable 100 Gbps rate) using 10 TCAMs (5 BPQs) each smaller than 110 Kb with access time 1.1 ns. A TCAM of this size has a rate of 900 millions accesses per second, and 3.5 accesses per operation (Insert or Dequeue) this TCAM-PPQ(3) works at a rate of 180 million packets per second (assuming some parallelism between Insert and Dequeue operations steps). Assuming average packet size of 140 bytes [13, 26], TCAM-PPQ(3) supports a line rate of 200 Gbps.

4. Power Sorting

We present the *PowerSort* algorithm (code is given in Appendix C), that sorts n items in $O(n)$ time using one BPQ with capacity $\sqrt{n}$. In order to sort n items, PowerSort considers the n items input as $\sqrt{n}$ sublists of size $\sqrt{n}$ each, and using the BPQ to sort each one of them apart (lines 3-13). Each sorted sublist is stored in a RList (see Section 3). Later on the $\sqrt{n}$ sublists are merged to one sorted list of n items (by calling PowerMerge on line 14). We use $PowerMerge_{s,t}$ to refer to the function responsible for the merging phase, this function merges a total of t keys divided to s ordered sublists using a BPQ with capacity s. The same BPQ previously used for sorting is used in the merge phase for managing the minimal unmerged keys one from each sublist, we call such keys local minimum of their sublists.

The merge phase starts by initialization of the BPQ with the smallest keys of the sublists (lines 17-20). From now on until all keys have been merged, we extract the smallest key in the list (line 23), put it in the output array, deletes it from the BPQ and insert a new one, taken from the corresponding sublist which the extracted key originally came from (line 27), i.e. this new key is the new local minimum in the sublist of the extracted key.

When running this algorithm with a RPQ, we can sort n items in $O(n)$ time requiring only $O(w \cdot \sqrt{n})$ TCAM entries. As can be seen from Section 4.2 these results are in some sense optimal.

4.1. The Power k Sorting

The PPQ(k) scheme can also be applied for the sorting problem. An immediate reduction is to insert all items to the queue and then dequeuing them one by one according to the sorted order. A more space efficient scheme can be obtained by using only one BPQ with capacity $\sqrt[k]{n}$ for all the functionalities of the $O(k)$ BPQs in the previous method. We use k phases, each phase $0 \le i < k$, starts with $n^{\frac{k-i}{k}}$ sorted sublists each contains $n^{\frac{i}{k}}$ items, and during the phase the BPQ is used to merge each $\sqrt[k]{n}$ of the sublists resulting with $n^{\frac{k-i-1}{k}}$ sorted sublists each with $n^{\frac{i+1}{k}}$. Therefore the last phase completes with one sorted list of n items.

This sorting scheme inserts and deletes each item k times from the BPQ (one time in every phase), therefore the time complexity remains $O(kn)$, but it uses only one BPQ. When using this method with TCAM based BPQ, this method will sort n items in $O(kn)$ TCAM accesses using $O(kw\sqrt[k]{n})$ TCAM space (in term of entries). This result matches the lower bound when a longest prefix TCAM is used, omitting the w factor in space and considering $r = \frac{1}{k}$ (see 4.2), thereby expressing its optimality. Similar to the TPQ(k) priority queue implementation, this sorting scheme presents an interesting time and TCAM space tradeoffs that can have big importance to TCAMs and scheduling systems designers.

4.2. Proving $\Omega(n)$ queries lower bound for TCAM sorting

Here we generalize Ben Amram's [27] lower bound and extend it to the TCAM assisted model. We consider a TCAM of size M as a black box, with a $query(v)$ - an operation that searches v in the TCAM resulting with one out of M possible outcomes, and a $write(p, i)$ - an operation that writes the pattern value p to the entry $0 \leq i < M$ in the TCAM but has no effect on the RAM.

Following [27], we use the same representation of a program as a tree in which each node is labeled with an instruction of the program. Instructions can be assignment, computation, indirect-addressing , decision and halt where we consider TCAM query as M outputs decision instruction and omit TCAM writes from the model. The proof of the next lemma is the same as in [27].

Lemma 4.1. *In the extended model, for any tree representation of a sorting program of n elements, the number of leafs is at least $n!$.*

Definition 4.2. An M,q-Almost-Binary-Tree ($ABTree_{M,q}$) *is a tree where the path from any leaf to the root contains at most q nodes with M sons each, the rest of the nodes along the path are binary (have only two sons).*

Lemma 4.3. *The maximal height of any $ABTree_{M,q}$ with N leafs is at least $\lfloor \log_2 N \rfloor - q\lceil \log_2 M \rceil$.*

Proof. we simply replace each M-node with a balanced binary tree of M leafs [4]. Each substitution adds at most $\lceil \log_2 M \rceil - 1$ nodes across all the paths from the root to any predecessor of the replaced M-node. In the resulting tree T', the maximal hight H' is at least $\log_2 N$. By the definition of q, at most $q \cdot (\lceil \log M \rceil - 1)$ nodes along the maximal path in T' are the result of nodes replacements. Therefore the maximal height H of the original tree T (before replacement) must satisfy:

$$H \geq H' - q\lceil \log M \rceil \geq \frac{n}{2} \log n - q\lceil \log M \rceil, \tag{5}$$

□

Theorem 4.4. *Any sorting algorithm that uses standard operators, polynomial size RAM and M size TCAMs, must use at least $\frac{n}{2} \log n - q \log M$ steps (in the worst case) to complete where q is the maximum number of TCAM queries per execution and n is the number of sorted items.*

Proof. Let T be the computation tree of the sorting algorithm as defined in [27], considering TCAM queries as M-nodes. A simple observation is that T is an $ABTree_{M,q}$ with at least $n!$ leafs. Therefore by Lemma 4.3 the maximal height of the tree is at least $\lfloor \log_2 n! \rfloor - q\lceil \log_2 M \rceil$. As $\log n! > \frac{n}{2} \log n$ we get that the worst case running time of the sorting algorithm is at least: $\frac{n}{2} \log n - q \log M$. □

Corollary 4.5. *Any $o(n \log n)$ time sorting algorithm that uses standard operators, polynomial size RAM and $O(n^r)$ size TCAMs, must use $\Omega(\frac{n}{r})$ TCAM queries.*

Proof. From Theorem 4.4, $\frac{n}{2} \log n - q \log M = o(n \log n)$, therefore

$$q = \Omega\left(\frac{n \log n}{\lceil \log M \rceil}\right).$$

By setting $M = O(n^r)$ we obtain that

$$q = \Omega\left(\frac{n}{r}\right).$$

□

[4]if M is not a power of 2 then the sub tree should be as balanced as possible

Corollary 4.6. *Any $o(n \log n)$ time sorting algorithm that uses standard operators, polynomial size RAM and $O(n^r)$ size BPQs, must use $\Omega(\frac{n}{r})$ BPQ operations.*

Proof. A BPQ of size $O(n^r)$ can be implemented with TCAMs of size $O(n^r)$ when considering TCAMs that return the most accurate matching line (the one with fewest '*'s). Such implementation performs O(1) TCAM accesses per operation, therefore, if there was a sorting algorithm that can sort n items using $O(n^r)$ size BPQs with $o(\frac{n}{r})$ BPQ operations then it was contradicting Corollary 4.5. □

Note that the model considered here matches the computation model used by the PPQ algorithm and also the implementation of the TCAM-PPQ. However one may consider a model that includes more CPU instructions such as shift-right and more, that are beyond the scope of our bound.

5. TCAM-PPQ Analytical Results

We compare our scheme TCAM-PPQ and TCAM-PPQ(3) to the optimized TCAM based PQ implementations RPQ, RPQ-2 and RPQ-CAO that are described in details in Appendix D. We calculate the required TCAM space and resulting packet throughput for varying number n of elements in the queue (i.e., n is the maximal number of concurrent flows). We set w, the key width to 36 bits which is above the minimum required in the current high end traffic demands.

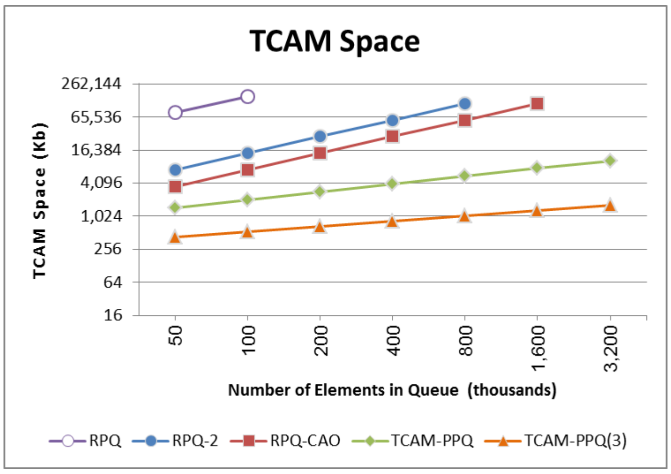


Figure 6: Total TCAM space (size) requirement for different number of elements PQ for the different implementation methods.

In Figure 6 we present the total TCAM space (over all TCAMs) required by each scheme. We assume that the TCAM chip size is limited to $72Mb$, which as far as we know is the largest TCAM available today [25]. Each of the lines in the graph is cut when the solution starts using infeasible TCAM building block sizes (i.e., larger than 72Mb). Clearly TCAM-PPQ and TCAM-PPQ(3) have a significant advantage over the other schemes since they require much smaller TCAM building blocks (and also total size) than the other solutions for the same PQ size. Moreover they are the only ones that use feasible TCAM size when constructing a one million elements PQ. All the other variations of RPQ require TCAM of size 1Gb for a million elements in the queue, which is infeasible in any aspect (TCAM price, or power consumption, or speed).

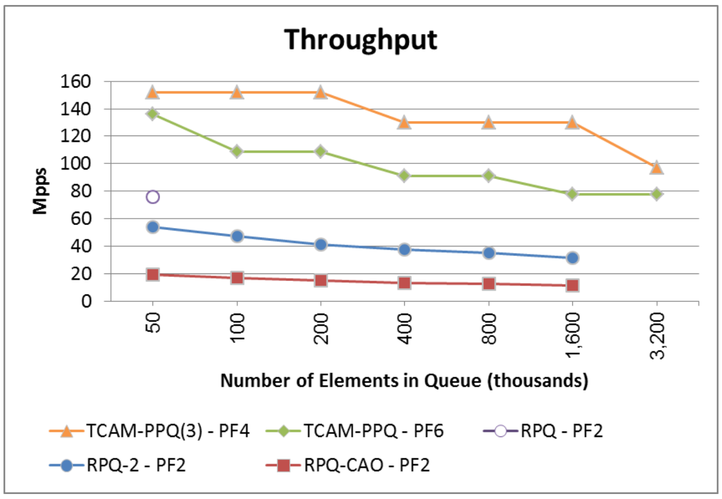


Figure 7: Packet throughput as a function of the number of elements. For each implementation we specify its Parallel Factor (PF) which stands for the maximal number of parallel accesses to different TCAMs.

In Figure 7 we present the potential packet throughput of the schemes in the worst case scenario. Similar to [25] and [28], we calculate the throughput considering only the TCAM accesses and not SRAM memory accesses. The rationale is that the TCAM accesses dominate the execution time and power consumption and it is performed in pipeline with the SRAM accesses. The TCAM access time is a function of the basic TCAM size. Recall that the TCAM speed increases considerably as its size reduces, [25, 28]. Next to each scheme we print the Parallelization Factor(PF), which is defined as the number of TCAM chips the scheme accesses in parallel. As can be seen in Figure 7, TCAM-PPQ and TCAM-PPQ (3) are the only schemes with reasonable throughput, of about 100Mpps for one millions timestamps, i.e., they can be used to construct a PQ working at a rate of 100Gbps. This is due to two major reasons: First, they use smaller TCAM chips and thus the TCAM is faster, and Secondly, have high Parallelization Factor and hence reducing the number of sequential accesses and thus increase the throughput. Note that the RPQ scheme achieves 75Mbps but it may be used with 50 elements, due to its high space requirement. Comparing TCAM-PPQ to TCAM-PPQ(3) we see that the latter is more space efficient and reach higher throughput levels. Table 1 summarizes the requirement of the different schemes.

In [15] a PQ design based on shift registers is presented which supports similar throughput as RPQ but cannot scale beyond 2048 items. By applying the PPQ scheme (results summarized in 8) we can extend it to hold one million items while supporting a throughput of 100 million packets per second as with TCAM-PPQ.

Method	Insert	Dequeue	space (#entries)
RPQ	2	1	$2w \cdot N$
RPQ-2	$\log w + 1$	1	$4N$
RPQ-CAO	$w/2 + 1$	1	$2N$
TCAM-PPQ	2	3	$6w \cdot \sqrt{N}$
TCAM-PPQ(3)	4	3	$10w \cdot \sqrt[3]{N}$

Table 1: Number of sequential TCAM accesses for the different TCAM based priority queues in an Insert and Dequeue operations (parallel access scheme is assumed).

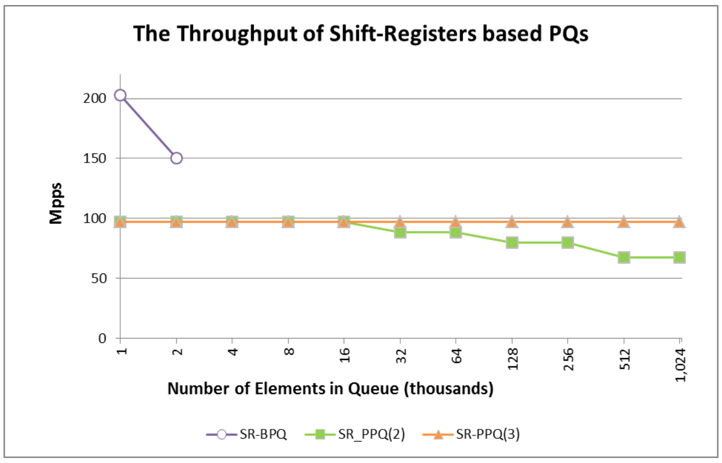


Figure 8: Packet throughput as a function of the number of elements.

6. Conclusions

This paper presents a sweet spot construction of a priority queue: a construction that enjoys the throughput and speed of small hardware priority queues without the size limitations they impose. It requires small hardware priority queues as building blocks of size cube root of the resulting priority queue size. We demonstrate the construction on the TCAM parallel technology, that when the size reduces works even faster. Combining these two together results in the first feasible and accurate solution to the packets scheduling problem while using commodity hardware. Thus we avoid the special, complex and inflexible ASIC design, and the alternative slow software solution (slow due to the inherent logarithmic complexity of the problem).

Our work shows that TCAMs can be used to solve a data structure problem more efficiently than it is possible in a software based system. This is another step in the direction of understanding the power of TCAMs and the way they can be used to solve basic computer science problems such as sorting and priority queuing.

References

References

[1] Y. Afek, A. Bremler-Barr, L. Schiff, Recursive design of hardware priority queues, in: Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '13, ACM, New York, NY, USA, 2013, pp. 23–32. doi:10.1145/2486159.2486194.
URL http://doi.acm.org/10.1145/2486159.2486194

[2] L. Kleinrock, A. Nilsson, On optimal scheduling algorithms for time-shared systems, Journal of the ACM 28 (3) (1981) 477–486.

[3] L. Kleinrock, R. Finklestein, Time dependent priority queues, Operations Research 15 (1) (1967) 104–116.

[4] L. Kleinrock, Queueing Systems: Volume 2: Computer Applications, John Wiley & Sons New York, 1976.

[5] L. Kleinrock, J. Hsu, A continuum of computer processor-sharing queueing models, in: Proceedings of the Seventh International Teletraffic Congress, Stockholm, Sweden, 1973, pp. 431/1–431/6.

[6] L. Zhang, Virtualclock: a new traffic control algorithm for packet-switched networks, ACM Transactions on Computer Systems (TOCS) 9 (2) (1991) 101 –124. doi:10.1145/103720.103721.

[7] P. Goyal, H. Vin, H. Cheng, Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks, Networking, IEEE/ACM Transactions on 5 (5) (1997) 690 –704. doi:10.1109/90.649569.

[8] S. Keshav, An engineering approach to computer networking: ATM networks, the Internet, and the telephone network, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[9] M. Thorup, Equivalence between priority queues and sorting, in: IEEE Symposium on Foundations of Computer Science, 2002, pp. 125–134. doi:10.1109/SFCS.2002.1181889.

[10] P. Lavoie, D. Haccoun, Y. Savaria, A systolic architecture for fast stack sequential decoders, Communications, IEEE Transactions on 42 (234) (1994) 324 –335. doi:10.1109/TCOMM.1994.577044.

[11] S.-W. Moon, K. Shin, J. Rexford, Scalable hardware priority queue architectures for high-speed packet switches, in: Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE, 1997, pp. 203 –212. doi:10.1109/RTTAS.1997.601359.

[12] H. Wang, B. Lin, Pipelined van emde boas tree: Algorithms, analysis, and applications, in: IEEE INFOCOM, 2007, pp. 2471–2475. doi:10.1109/INFCOM.2007.303.

[13] K. Mclaughlin, S. Sezer, H. Blume, X. Yang, F. Kupzog, T. G. Noll, A scalable packet sorting circuit for high-speed wfq packet scheduling, IEEE Transactions on Very Large Scale Integration Systems 16 (2008) 781–791. doi:10.1109/TVLSI.2008.2000323.

[14] A. Ioannou, M. Katevenis, Pipelined heap (priority queue) management for advanced scheduling in high-speed networks, Networking, IEEE/ACM Transactions on 15 (2) (2007) 450 –461. doi:10.1109/TNET.2007.892882.

[15] R. Chandra, O. Sinnen, Improving application performance with hardware data structures, in: Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, 2010, pp. 1 –4. doi:10.1109/IPDPSW.2010.5470740.

[16] R. Panigrahy, S. Sharma, Sorting and searching using ternary cams, IEEE Micro 23 (2003) 44–53. doi:10.1109/MM.2003.1179897.

[17] A. Kortebi, L. Muscariello, S. Oueslati, J. Roberts, Evaluating the number of active flows in a scheduler realizing fair statistical bandwidth sharing, in: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, SIGMETRICS '05, ACM, New York, NY, USA, 2005, pp. 217–228. doi:http://doi.acm.org/10.1145/1064212.1064237.
URL http://doi.acm.org/10.1145/1064212.1064237

[18] M. Shreedhar, G. Varghese, Efficient fair queueing using deficit round-robin, IEEE/ACM Trans. Netw. 4 (1996) 375–385. doi:http://dx.doi.org/10.1109/90.502236.
URL http://dx.doi.org/10.1109/90.502236

[19] H. Wang, B. Lin, Succinct priority indexing structures for the management of large priority queues, in: Quality of Service, 2009. IWQoS. 17th International Workshop on, 2009, pp. 1 –5. doi:10.1109/IWQoS.2009.5201416.

[20] X. Zhuang, S. Pande, A scalable priority queue architecture for high speed network processing, in: INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings, 2006, pp. 1 –12. doi:10.1109/INFOCOM.2006.137.

[21] G. S. Brodal, J. L. Trff, C. D. Zaroliagis, A parallel priority queue with constant time operations, Journal of Parallel and Distributed Computing 49 (1) (1998) 4 – 21. doi:10.1006/jpdc.1998.1425.

[22] A. V. Gerbessiotis, C. J. Siniolakis, Architecture independent parallel selection with applications to parallel priority queues, Theoretical Computer Science 301 (13) (2003) 119 – 142. doi:10.1016/S0304-3975(02)00572-8.

[23] J. Garcia, M. March, L. Cerda, J. Corbal, M. Valero, On the design of hybrid dram/sram memory schemes for fast packet buffers, in: High Performance Switching and Routing, 2004. HPSR. 2004 Workshop on, 2004, pp. 15 – 19. doi:10.1109/HPSR.2004.1303414.

[24] H. J. Chao, B. Liu, High Performance Switches and Routers, John Wiley & Sons, Inc., 2006.

[25] J. Patel, E. Norige, E. Torng, A. X. Liu, Fast regular expression matching using small tcams for network intrusion detection and prevention systems, in: USENIX Security Symposium, 2010, pp. 111–126.

[26] CAIDA, Packet size distribution comparison between Internet links in 1998 and 2008.
URL http://www.caida.org/research/traffic-analysis/pkt_size_distribution/graphs.xml

[27] A. M. Ben-amram, When can we sort in o(n log n) time?, Journal of Computer and System Sciences 54 (1997) 345–370.

[28] B. Agrawal, T. Sherwood, Ternary cam power and delay model: Extensions and uses, IEEE Transactions on Very Large Scale Integration Systems 16 (2008) 554–564. doi:10.1109/TVLSI.2008.917538.

[29] D. Shah, P. Gupta, Fast updating algorithms for tcams, IEEE Micro 21 (2001) 36–47. doi:10.1109/40.903060.

[30] Intel 64 and IA-32 Architectures Software Developer's Manual: Vol. 2A.
URL http://www.intel.com/

[31] Software Optimization Guide for AMD Family 10h and 12h Processors.
URL http://www.amd.com/

Appendix A. The PPQ algorithm

```
 1: function PPQ.INIT($n$)
 2:     $in \leftarrow 0$
 3:     $out \leftarrow 1$
 4:     input-BPQ[$in$] ← new BPQ ( $\sqrt{n}$)
 5:     input-BPQ[$out$] ← new BPQ ( $\sqrt{n}$)
 6:     exit-BPQ ← new BPQ ( $\sqrt{n}$)
 7:     buffer[$in$] ← new RList ( $\sqrt{n}$)
 8:     buffer[$out$] ← new RList ( $\sqrt{n}$)
 9:     small-sublists ← new RList ( $\sqrt{n}$)
10:     fused-sublist ← null
11: end function

12: function BACKGROUND
13:     Do 2 steps in merging buffer[$in$] with fused-sublist    ▷ fused-sublist is merged with
    buffer[$in$], both are in the SRAM; In this step two merge steps are performed.
14:     if input-BPQ[$out$].count > 0 then
15:         item ← input-BPQ[$out$].Dequeue()
16:         buffer[$out$].Push(item)
17:     end if
18: end function

19: function PPQ.INSERT(item)
20:     if input-BPQ[$in$].count = $\sqrt{N}$ then
                                                          ▷ A new full list is ready
21:         swap $in$ with $out$
22:         fused-sublist ← small-sublists.Pop()
23:         input-BPQ[$in$].Insert (item)
24:         Background()
```

```
25:         if fused-sublist.head > buffer[in].head then
    ▷ Need to replace the head item of fused-sublist which is in the exit-BPQ, head of buffer[in]
    is going to be the new head of fused-sulist
26:             exit-BPQ.Delete(fused-sublist.head)
27:             exit-BPQ.Insert (buffer[in].head)
28:         end if
29:     else
30:         Background()
31:         input-BPQ[in].Insert (item)
32:     end if
33: end function

34: function PPQ.DEQUEUE
35:     min1 ← min(input-BPQ[in].Min, buffer[out].Min)
36:     if exit-BPQ.Min < min1  then
37:         min ← exit-BPQ.Dequeue()
38:         remove min from min.sublist
                                        ▷ min.sublist is the RList that contained min.
39:         local-min ← new head of min.sublist
40:         exit-BPQ.Insert (local-min)
41:         if min.sublist.count = $\sqrt{N}$ then
42:             small-sublists.Push(min.sublist)
43:         end if
44:     else
45:         if input-BPQ[in].min < buffer[out].head then
46:             min ← input-BPQ[in].Dequeue()
47:         else
48:             min ← buffer[out].Pop()
49:         end if
50:     end if
51:     Background()
52:     return min
53: end function
```

Appendix B. Reducing the worst case number of BPQ accesses in a PPQ.insert operation from 3 to 2

In this appendix we explain how to reduce the worst case number of BPQ accesses in a PPQ.insert operation from 3 to 2. A careful look at the PPQ.insert algorithm reveals that only once every $\sqrt{n}$, when the input-BPQ is exactly full may this operation require 3 sequential accesses, in all other cases this operation requires only 1 sequential access. It requires 3 operation if the head of the buffer[*in*] is smaller than the head of the sublist marked to be merge with it (the fused-list in code). This 3 sequential accesses consist of Insert to the input-BPQ and Delete and Insert to the exit-BPQ, can be broken by delaying the last access in the sequence (line 27) to the next Insert operation. Notice that now each dequeue operation needs to check whether the minimum that needs to be returned is this delayed value, as in the pseudo-code below. Implementing this delay requires the following changes to the algorithm:

- Delaying the insert (in line 27) - the existing line should be replaced by:

```
1: wait-head ← new-sublist.head
```

- Performing delayed insertion - the following code should be added just before line 31:

```
1: if wait-head ≠ null then
2:     exit-BPQ.Insert(wait-head)
3:     wait-head ← null
4: end if
```

- Check if delayed item should be dequeued - we need to ensure that Insert() doesn't miss the minimum item when it is the delayed new-sublist head. By comparing the delayed head to other minimums the Dequeue can decide whether it should be used. This change is implemented by adding the following lines at the beginning of Dequeue:

```
1: if wait-head ≠ null then
2:     if wait-head < input-BPQ.min &&
3: wait-head < merge-list.min then
4:         min ← wait-head
5:         remove wait-head from wait-head.sublist
6:         local-min ← new head of wait-head.sublist
7:         exit-BPQ.Insert(local-min)
8:         wait-head ← null
9:         Background()
10:        return min
11:    end if
12: end if
```

Appendix C. The Power Sorting Scheme

```
1: function PowerSort(Array In, List Out, $n$)
2:     q ← new BPQ ( $\sqrt{n}$)
3:     for $i = 0$ to $\sqrt{n} - 1$ do
4:         for $j = 0$ to $\sqrt{n} - 1$ do
5:             q.Insert(In[$i \cdot \sqrt{n} + j$)
6:         end for
7:         Subs[$i$] ← new RList ( $\sqrt{n}$)
8:         for $j = 0$ to $\sqrt{n} - 1$ do
9:             item ← q.Dequeue()
10:            item.origin-id ← $i$
11:            Subs[$i$].Push(item)
12:        end for
13:    end for
14:    PowerMerge(Subs, Out, q, $\sqrt{n}$, $\sqrt{n}$)
15: end function

16: function PowerMerge(RList Subs[], RList Out, BPQ q, $s$, $t$)
17:     for $i = 0$ to $s$ do                    ▷ $s$ is the number of sublists
18:         local-min ← Subs[$i$].Pop()
```

```
19:         q.Insert(local-min)
20:     end for
21:     count ← 0
22:     for count = 1 to t do                          ▷ t is the total num. of items
23:         min ← q.Dequeue()
24:         id ← min.origin-id
25:         if Subs[id] not empty then
26:             local-min ← Subs[id].Pop()
27:             q.Insert(local-min)
28:         end if
29:         Out.Push(min)
30:     end for
31: end function
```

Appendix D. The TCAM Ranges based PQ (RPQ) as Building Block

Here we construct the basic building block used in all the TCAM based PQ's constructions, TCAM-PPQ, TCAM-PPQ(3), and TCAM-PPQ(k). The RPQ in a simpler form was suggested in [16] which in turn is based on a TCAM based set of ranges data structure, which is the first building block described below.

Appendix D.1. Implementing a Set of Ranges with TCAMs

A set of ranges contains range items of the form $[a, b]$, where a and b are integers, $a \leq b$. The set supports three operations: the insertion of a new range, deletion of an existing range and a lookup for the range that contains a given point. Only sets of disjoint ranges are considered here.

Panigrahy and Sharma have shown [16] that we can manage a set of ranges using two TCAM entries per range, and two TCAM queries per point lookup. They named their solution PIDR (Point Intersection Disjoint Ranges). Following [16] we define:

- *The Longest Common Prefix (LCP) of integers a and b (represented by w bits)* is the longest prefix shared by the binary representations of a and b. For example, LCP(a = 0101,b = 0111) = 01.

- *The Extended LCP (ELCP) patterns of integers a and b, named* 0*-ELCP and* 1*-ELCP* are the patterns that extend the LCP of a and b by one more bit (0 and 1) and use '$*$'s' for the remaining bits, for example, let a = 0101,and b = 0111 then 0-ELCP(a, b)=010$*$, and 1-ELCP(a, b) = 011$*$, (w = 4).

The set of ranges is maintained by keeping for each range $[a, b]$ its two ELCP patterns in two separate TCAMs, one for the 0-ELCP's and one for the 1-ELCP's. The ELCP patterns, that stored in the TCAMs, are associated with range objects that located on SRAM. It is easy to see that if $x \in [a, b]$ then x matches exactly one of the ELCP patterns of $[a, b]$. But the opposite is not always true, x might match an ELCP of a range that does not contain x.

Thus if $x \in I = [a, b]$, then the binary presentation of x matches either 0-ELCP(a, b), or 1-ELCP(a, b). Moreover, while x might match other ELCP's, the a, b ELCP it matches is the longest one, i.e., with the fewest number of $*$'s as proved in [16]. Therefore, if the ELCP's are stored in each TCAM in order of increasing number of ”don’t care”s ($*$’s) (as for example in Figure D.10) then a TCAM that returns the first entry with a pattern that matches x returns either the 0-ELCP or the 1-ELCP of I.

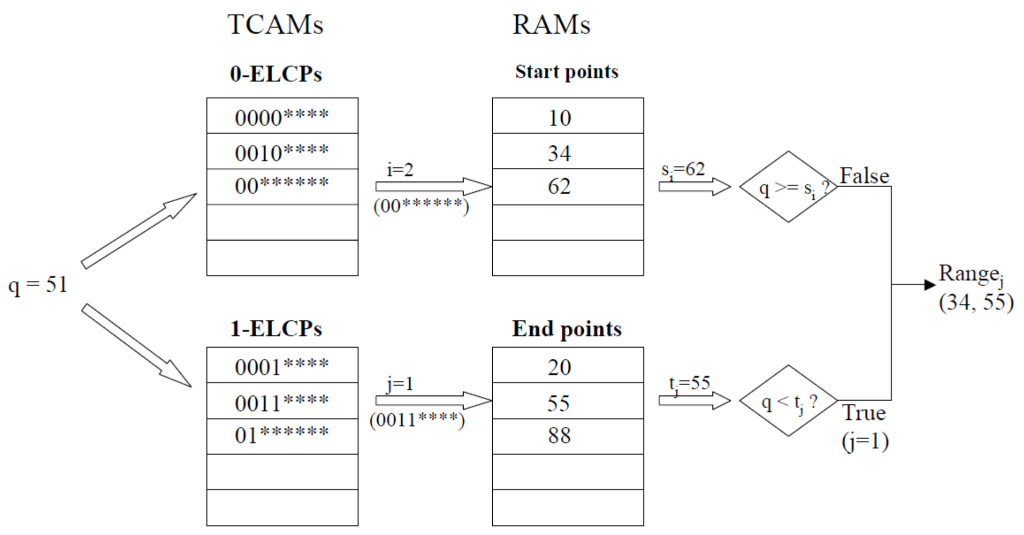


Figure D.9: (Figure 3 in [16]) The set contains the ranges $\{[10, 20], [34, 55], [62, 88]\}$ and is searched for the range that contains the value 51.

The algorithm for range search, as described in Figure D.9 (taken from [16]) is simple, we query both TCAMs; then we check if one of the matches belongs to a range that contains the point. The remaining challenge is then to maintain the ELCP’s in the TCAM so that the longest prefix match is returned. One way to guarantee it is to use a longest prefix match TCAM, but this kind of TCAM is too expensive and too slow in large sizes. Panigrahy and Sharma [16] suggest three principal methods to guarantee longest prefix match with standard TCAM, we name them *PIDR-1*, *PIDR-2* and *PIDR-CAO*.

- The *PIDR-1* method uses $(w \cdot m)$ entries TCAM to hold m patterns, where entries jm to $(j + 1)m - 1$, $j = 0, \ldots w - 1$ may hold only patterns with j $*$’s. This approach yields an $O(1)$ accesses per operation in the expense of a factor w in the TCAM size (see Figure D.10). The actual number of entries is $(w - \log m + 2)m$, which is smaller than wm, since at the bottom part there are less than m possible patterns of length smaller than $\log m$.

- The *PIDR-2* method uses only m entries each of length $2w$ holding the pattern and an unary presentation of the corresponding prefix length. For example let $w = 5$, $011 * *| * *1 * *$ where $**1**$ represents the number 3. In general a pattern with prefix length l is appended with the string $*^{l-1}1*^{w-l}$. This way of presentation enables binary search of the most longest matching pattern. For Example, to match patterns of length between l_1 and l_2, where $l_1 \leq l_2$, one should attach to the queried value q the pattern $0^{l_1-1}1^{l_2-l_1+1}0^{w-l_2}$. This approach finds the longest matching prefix in $\log w$ accesses and use one access per update.

- The *PIDR-CAO* stores the patterns in m entries TCAM (of regular w bits) while ensuring chain ancestor order (if pattern p_1 is the prefix of pattern p_2 then p_1 is stored bellow p_2) using TCAM memory management scheme called CAO[29]. The idea behind CAO is to manage a pool of free entries in the middle of the TCAM and ensuring that each increasing chain of patterns (each one is the prefix of the next) will be divided between the two filled regions of the TCAM (above and below the free pool), thereby minimizing the amount of TCAM updates involved with each pattern insertion. This scheme is reported to have one TCAM update per insert/delete but this result was achieved while checking updates to IP-Lookup tables (that also requires longest prefix order) and there is no reason that this result would also applies to our case and a worst case of w/2 updates should be considered (which means w for two TCAMs). Thus, due to this high worst case overhead, this solution cannot be used in high throughput environments either.

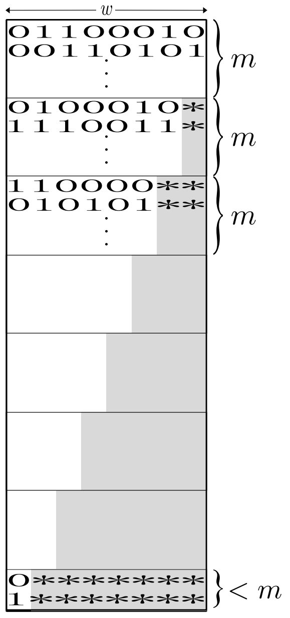


Figure D.10: TCAM naive memory management for m patterns of width w, using $O(mw)$ entries. In the TCAM-PPQ scheme we use $m = \sqrt{n}$.

Appendix D.2. Implementing a Sorted List with TCAMs

Following [16] a sorted list of items $a_1 < a_2, ..., < a_m$ is maintained as a set of ranges as follows: $[0, a_1-1]$,$[a_1, a_2-1]$,. . .,$[a_m, 2^w-1]$. In this set the ELCP patterns of the ranges are stored in the TCAMs and associated with the range objects that are located in SRAM, each range object is associated with one pattern in the 0-ELCPs TCAM and one in 1-ELCPs TCAM. However, to be able to extract items by order, we keep the ranges in the set connected to each other forming a doubly linked list (see Figure D.11), ordered by their values, i.e., range $[a_i, a_{i+1} - 1]$ is linked in both directions with range $[a_{i+1}, a_{i+2} - 1]$. Each range object is also linked with the items matching its min or max values (also called the endpoints of the range).

Following the last discussion, each range object contains the values of its endpoints, the offsets of its ELCP patterns in the TCAMs, pointers to adjacent range objects and pointers to items with key value matching one of its endpoints. Moreover, for each TCAM we use equal sized SRAM array which allows to translate a query result (the offset of matched pattern) to the relevant range object. We also extend each item with a pointer to his containing range so we can save a TCAM query when deleting an item from the sorted list.

We refer to this implementation by *Ranges based Priority Queue (RPQ)* when it is based on PIDR-1, considering it the best candidate for our powering technique. Otherwise we refer to this implementation by RPQ-2 or RPQ-CAO when based on PIDR-2 or PIDR-CAO, respectively.

In Appendix E we provide RPQ optimizations and implementation details such as management of TCAM free entries, patterns generation and priorities wrap around. Using these optimizations and parallel execution on both TCAMs (the 0-ELCP and 1-ELCP) the cost of a RPQ is as described in Table D.2.

Operation	Updates	Queries	Comment
Insert(item)	1	1	only 1 update for empty list
Delete(item)	3	0	item is linked to a range
Dequeue()	1	0	

Table D.2: RPQ optimization results, with parallel execution on both TCAMs.

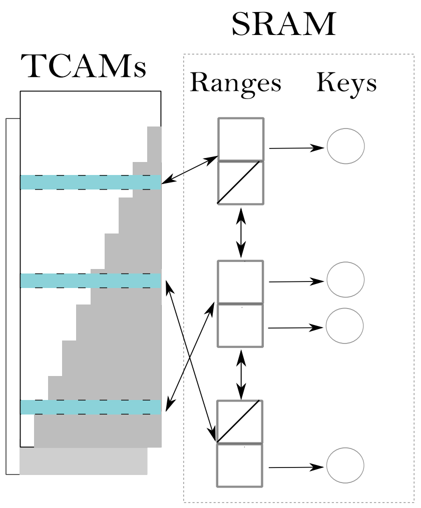


Figure D.11: RPQ implementation. Only associated TCAM entries (full marked lines) are occupied (the other entries are currently empty).

Appendix E. Optimizing RPQ

Insertion and deletion from RPQ are recurring operations in the TCAM-PPQ (2-3 times per packet), therefore decreasing their cost improves the overall performance of TCAM-PPQ. First reduction, also implied by [16], requires that when Insert(item) splits a range, it should split it in a way that keeps the ELCPs of the range in the TCAMs and reusing them for one of the resulting sub-ranges. In this way we save 4 TCAM updates, two for deleting the old patterns (of the splitted range) and two for inserting new patterns (for one of the sub-ranges). This reduction is possible by the following observation: for each range $[a, b]$ and splitting key x, in one of the possible splits $[a, x - 1]$ and $[x, b]$ or to $[a, x]$ and $[x + 1, b]$, one of the resulting sub-ranges, share the same ELCP patterns as the original. Therefore in order to make a split we only add a second range and update the standard memory location that specifies the edge values of the original range.

Note that this insert mechanism can create empty ranges (ranges that are not occupied by items), for example suppose we have the range $[3, 10]$ where only 3 is associated with an item, when we insert 8, the ELCPs of $[3, 8]$ are the same are those of $[3, 10]$, therefore the outcome sub-ranges will be $[3, 8]$ and $[9, 10]$, where $[9, 10]$ is empty. Empty ranges are not a desirable outcome because they consume TCAM space but don't represent any item, thereby decreasing the list capacity.

To bound the number of empty ranges, we require the delete mechanism to try and merge the range (of the deleted item) with one of its neighbor ranges even if only one of his edges is not occupied (and not both as implied in Section Appendix D.2). This simple extension can ensure that any two adjacent range edges that doesn't belong to the same range can't be both unoccupied, this is valid since when Insert splits it makes one of the edges occupied and when Delete make an edge unoccupied it merges it with a neighbor unless the adjacent edge is occupied. Using the fact that at least one of every two adjacent edges is occupied, we get that the number of occupied edges is at least half of the number of edges which means that the number of items is at least the number of ranges.

The cost of the improved delete mechanism is 3 (when operated parallely on both TCAMs), since in the worst case we need to delete two ranges and add new merged range. It turns out that we can disable the merging in DeleteMin and only delete the range if it remains empty, thereby reducing its cost to 1. Note that this requires a small change in Insert to merge ranges in the case an insertion is made to the beginning of the list. The results of these optimizations are summarized in Table D.2.

Appendix E.1. Managing TCAM free entries

When we write a new pattern to a TCAM we are required to find an appropriate free entry. By the naive TCAM management approach, the prefix length of the pattern indicates a region of size $\sqrt{n}$ allocated for entries of that length, but we still need to find a specific free entry within this region. To do so we manage a pool of free entries per each region. Our pool is implemented by a counter combined with a linked list. When an entry is requested it is extracted from the linked list or if the list is empty the counter value is returned and increased. When an entry is cleaned its offset is added to the linked list and when the list is reset the linked list is initialized by empty list and the counter is set to zero. This implementation allows a $O(1)$ reset required by the sorting list and reusing of cleaned entries required especially by the merge list which is never reset.

Appendix E.2. Computing ELCP

Writing a pattern to a TCAM, involves the creation of a mask word with bit '1' for every "don't care" character in the pattern, and '0' elsewhere. Therefore in order to write the

ELCP of range $[a, b]$ one should compute the mask value $(1 << \lceil \log_2(a \oplus b) \rceil) - 1$. Computing $\lceil \log_2(a \oplus b) \rceil$, i.e the most significant bit (msb) of $a \oplus b$, for 64 bit numbers can be done with Bit Scan Reverse (BSR) instruction provided by modern 64-bit CPUs [30, 31]. When $w > 64$, the offset of the msb can be founded by using one query to a TCAM with w entries.

Appendix E.3. Handling priorities' wrap around

Usually the range of priority values used in the PQ is much smaller than the real range of timestamps used in the scheduling system and therefore priority values may wrap around. For example, timestamps that measure packet transmission times in resolution of byte in 100Gb/s rate, will increment by $12.5 \cdot 10^9$ per second making a 32bit priority key to wrap around 3 times per second. Inserting both pre and post wrap around timestamps to the PQ will result with order distortion (post wrap around timestamps will be regarded as smallest). Overcoming the wrap around is possible assuming that the active timestamps values are bounded in a range of size 2^w, where w is the size, in bits, of priority values. The solution we suggest is to use two PQs, that share the same TCAM resources of a single TCAM based PQ of size n. One PQ stores elements that originate from timestamps with '0' as their $w + 1$ most significant (MSB) bit, and the second PQ stores the rest (those with '1' as their $w + 1$ MSB bit). GetMin uses a global precedence-flag to decide which PQ has precedence over the other (holds the smaller elements). This flag is flipped at each wrap around. To allow the two PQs to share the same TCAMS, an extra bit is appended to each TCAM entry. The total operation time remains almost the same except for the flag check and/or flip.

Note that this solution can also be applied to a multi queues scenario by making the precedence-flag global and shared among all the queues in the system.