BBQ: A Fast and Scalable Integer Priority Queue for Hardware Packet Scheduling

Nirav Atre, Hugo Sadok, Justine Sherry
Carnegie Mellon University

Abstract

The need for fairness, strong isolation, and fine-grained control over network traffic in multi-tenant cloud settings has engendered a rich literature on packet scheduling in switches and programmable hardware. Recent proposals for hardware scheduling primitives (e.g., PIFO, PIEO, BMW-Tree) have enabled run-time programmable packet schedulers, considerably expanding the suite of scheduling policies that can be applied to network traffic. However, no existing solution can be practically *deployed* on modern switches and NICs because they either do not scale to the number of elements required by these devices or fail to deliver good throughput, thus requiring an impractical number of replicas.

In this work, we ask: is it possible to achieve priority packet scheduling at line-rate while supporting a large number of flows? Our key insight is to leverage a scheduling primitive used previously in software – called Hierarchical Find First Set – and port this to a highly pipeline-parallel hardware design. We present the architecture and implementation of the Bitmapped Bucket Queue (BBQ), a hardware-based integer priority queue that supports a wide range of scheduling policies (via a PIFO-like abstraction). BBQ, for the first time, supports hundreds of thousands of concurrent flows while guaranteeing 100 Gbps line rate (148.8 Mpps) on FPGAs and 1 Tbps (1,488 Mpps) line rate on ASICs. We demonstrate this by implementing BBQ on a commodity FPGA where it is capable of supporting over 100K flows and 32K priorities at 300 MHz, 3× the packet rate of similar hardware priority queue designs. On ASIC, we can synthesize 100K elements at 3.1 GHz using a 7nm process.

1 Introduction

Packet scheduling, the problem of deciding what *order* and/or *time* network packets ought to be served or transmitted, has long occupied a prominent position in networking literature. Prior work in this space has resulted in a rich repertoire of packet scheduling algorithms with a variety of different properties: fairness and starvation avoidance [9], attack resilience [7], burst reduction [34], optimal flow completion time [4], *etc.* At the heart of many of these algorithms is a *priority queue* data-structure, which allows the scheduler to map packets' relative order (or scheduling time) to unique priorities, sort them, and subsequently extract the highest-priority item (i.e., the next packet to schedule) from the queue.

Despite these strong theoretical foundations, network switches and NICs have historically failed to provide anything beyond a small suite of simple scheduling algorithms [41]. The key problem is the *complexity associated with implementing a fast, scalable, and generic priority queue in hardware*. PIFO [41] was a foundational paper in articulating the importance of priority queueing and providing a practical implementation of a hardware priority queue. Yet, as we will discuss further in §2, PIFO falls short of achieving suitable performance for deployment either on an ASIC (as in a traditional switch) or on an FPGA (as in modern SmartNICs). Today, line rates run at 100+ Gbps, and comprise tens or even hundreds of thousands of concurrent flows, but PIFO can support at most 2048 concurrent flows while guaranteeing line-rate processing.

This performance bottleneck arises because implementing an exact priority queue involves comparison-based sorting a sequence of n arbitrary elements, which imposes a theoretical lower bound of $\Omega(n \log n)$ on the required number of comparator operations. While PIFO reduces this complexity to $O(n)$ by observing that priority can be determined at enqueue time, and the spatial nature of switch or NIC hardware allows this computation to be parallelized (i.e., by unrolling the operations in *space* instead of *time*), spatial parallelism only scales so far: as the number of comparator operations increases, so does the complexity of the resulting datapath circuit, causing the maximum operating frequency of the switch or NIC to drop dramatically. Although prior work has since improved upon PIFO using more sophisticated parallelization techniques [40, 47]), they are nonetheless limited in terms of performance, scalability, or both. Moreover, many of these solutions give up logical partitioning, a key feature of PIFO that is essential for practical switch deployment. For instance, the state-of-the-art priority queue design, BMW-Tree [47], would require 1,056 replicas in order to support a 32×400 Gbps output-queued switch (§2.1.1)!

In this paper, we ask: *is it possible to achieve priority packet scheduling on NICs and switches at state-of-the-art line rates with 100K+ flows without sacrificing accuracy or the expressiveness that PIFO offers?*

As we look to build a highly performant and scalable priority queue for hardware packet scheduling, we are inspired by data structures with constant worst-case time complexity. In particular, we find that *integer priority queues* (IPQs) are well suited to the task: they subvert the complexity barrier imposed by comparison-based sorting, so performance and scalability are no longer at odds. Further, there should be no loss in precision so long as the relative ordering of packets can be encoded in the priority span of the IPQ.

In this work, we present the design and implementation

of the *Bitmapped Bucket Queue* (BBQ),[1] a scalable, high-performance IPQ for hardware packet scheduling. The data structure underlying BBQ is a Hierarchical Find-First Set (HFFS) queue [38, 45], which guarantees constant worst-case time complexity for standard heap operations. While HFFS is a well-known data structure that has found applications in several software systems,[2] our key insight is that *HFFS queueing is amenable to a highly efficient, fully-pipelined hardware implementation*. As a consequence, BBQ enables, for the first time, packet scheduling at 100 Gbps line rate using minimum-sized packets on a commodity FPGA-based SmartNIC, and can be incorporated into state-of-the-art 32 × 400 Gbps switches [33] with as little as 12 replicas.

Although efficient pipeline parallelism is key to BBQ's high performance and scalability, incorporating this parallelism introduced new challenges in avoiding data hazards (parallel reads and writes to the same data) and correctness. As we will discuss in §4.2, hazards manifested in our design in three ways and required careful design to disentangle parallel access to shared data *without* sacrificing performance. Nonetheless, BBQ ultimately makes one sacrifice to correctness in that it cannot support back-to-back dequeues of packets from the same flow; in §5 we show that by combining BBQ with a tiny PIFO instance we can guarantee the *performance* of BBQ with the *correctness* of PIFO.

The rest of this paper is organized as follows. In §2 we provide background and discuss the challenges of incorporating a programmable scheduler into modern switches and NICs. In §3, we provide an overview of the BBQ design, followed by a more detailed description of the architecture in §4, focusing on the challenges due to parallelism and hazards. In §5, we explore an augmentation to the BBQ design that counteracts the latency artifacts introduced by pipelining. In §6 we evaluate BBQ, followed by a discussion on how to incorporate the design into modern switches and NICs in §7. We then describe the related work in §8, discuss limitations and future work in §9, and conclude in §10.

2 Background and Motivation

More than ever, there is a need for programmable packet scheduling in hardware. Switches have long offered a limited set of packet scheduling algorithms and NICs are increasingly taking over dataplane tasks traditionally performed in software [37], including end-host packet scheduling [26, 34, 42]. In the case of switches, having a programmable packet scheduler could not only vastly expand the catalog of scheduling algorithms available to network operators, but also pave the way for faster innovation and customization [11, 41]. With NICs, system administrators already *expect* to be able to customize the packet scheduler that runs on the end host [37].

[1] Available at https://github.com/cmu-snap/BBQ.

[2] The Linux kernel uses FFS-based priority queueing for process scheduling. Eiffel [38] demonstrates how HFFS queueing can be used to realize high-performance *software* packet scheduling, and [45] even gives a sketch for a hardware design.

2.1 Lack of Support for Emerging Use-Cases

PIFO [41] is a seminal work in this regard. It makes the observation that, when considering a single node [28], all scheduling algorithms can be expressed in how they make two decisions: *which* packet to schedule next and *when* to schedule it. The authors observe that for many scheduling algorithms this behavior can be captured simply with a hardware priority queue. This priority queue can be used in one of two ways: it can implement work conserving algorithms by sorting flows by rank, or it can implement non-working conserving algorithms by sorting flows by scheduling time. While PIFO was initially conceived to run on switches, it has also been shown to be a useful primitive to schedule packets on NICs [26, 42].

Unfortunately, existing solutions for programmable packet scheduling in hardware fail to meet the requirements for both state-of-the-art NICs and switches. In what follows, we elaborate on these two use cases, highlighting how their stringent performance requirements are at odds with existing proposals for programmable hardware packet schedulers.

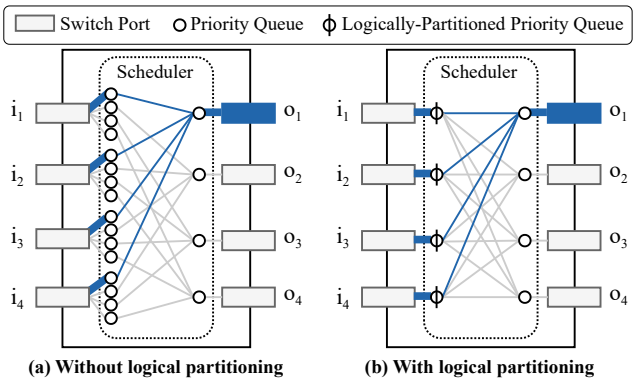
2.1.1 Line-Rate Switches

In order to support packet scheduling without impeding the rest of the switch's dataplane functionality, any realistic proposal for a packet scheduler ought to satisfy two key requirements:

(A1) Match the switch's aggregated packet rate: For scheduling in output-queued switches, the worst-case throughput demand corresponds to the scenario where ingress traffic from all ports is incident on a single egress port (i.e., incast behavior). In order to avoid backpressuring the switch fabric, the packet scheduler must be able to process packets at the same rate as the switch backplane (i.e., its aggregated packet rate) at *any* given port. For instance, in NVIDIA's Spectrum SN4700 (a state-of-the-art 400 GbE switch with 32 ports), the packet scheduler for a single port must be able to handle an aggregated packet rate of 8.4 *billion* pps [33].

(A2) Allow every port to address all buffered packets: In order to efficiently utilize on-chip memory, switches use a shared packet buffer that is dynamically partitioned between its output ports [41]. In the worst case, every packet in the shared buffer might be destined for the same output port, and each packet might map to a different flow to be scheduled. If the packet scheduler at a given port could only address a subset of buffered elements, it would impose additional constraints on the switch's ability to handle such bursts. Thus, a second key requirement for schedulers is the ability to allow any output port to address *every* buffered packet. In modern switches, shared packet buffers are provisioned for *hundreds of thousands* of packets [33].

In 2016, a single physical PIFO instance could handle the full aggregated packet rate for a 64-port 10 GbE switch (1 Bpps) [41]. Today, state-of-the-art switches, *e.g.* the SN4700, offer 400 GbE line rates with 32 ports (20× higher



(a) Without logical partitioning **(b) With logical partitioning**

Figure 1: Scheduler architecture using priority queues without (left), and with (right) support for logical partitioning. In a k-port switch using priority queues without logical partitioning, we need k^2+k priority queues in order to both implement output queueing *and* absorb incast traffic from all the ports. Support for logical partitioning reduces the number to $2k$.

aggregated throughput). Given the large (and ever-widening) gap between network and processing speeds, a single priority queue instance can no longer sustain aggregated packet rates. Consequently, to satisfy (A1), packet schedulers for modern switches must "scale out" using a mesh of priority queues.

Unfortunately, *priority queue designs that do not support logical partitioning, i.e., the ability to multiplex several independent queues atop a single physical queue, require prohibitively large meshes.* To the best of our knowledge, PIFO and PIEO are the only existing design that support logical partitioning, while more recent proposals (*e.g.*, BMW-Tree [47]) do not. Generously assuming that a single priority queue instance could handle 400 Gbps line-rate input in a k-port switch, these designs would require, at minimum, a (k^2+k) mesh of instances to realize per-output-port scheduling while supporting the full aggregated throughput due to incasts; an example mesh for $k=4$ is depicted in Figure 1(a). Further, *every instance would have to be provisioned with hundreds of thousands of queue elements* to satisfy (A2). Overall, a 32-port switch operating at 400 Gbps line-rate would require at least 32^2+32 BMW-RPU instances, each provisioned with 100K+ entries, to implement priority queueing alone. Based on synthesis numbers reported by the authors and considering that a switch chip area ranges from 200 mm^2 to 800 mm^2 [41] this corresponds to $1.5-6\times$ the total area.

The ability to logically partition a physical priority queue enables a significantly simpler mesh architecture. Assuming, again, that each priority queue instance can sustain 400 Gbps input, we would only need $2k$ instances, as depicted in Figure 1(b). Each physical instance in the first layer ingests packets from a single ingress port, but enqueues into one of k *logically independent queues* corresponding to the k possible destination ports. The second layer then periodically schedules packets among their k inputs, feeding traffic to their respective egress ports. *Despite the fact that PIFO benefits from this architecture, its inability to scale beyond 2,048 elements means that it does not satisfy (A2). Conversely, while PIEO scales marginally better, it does not provide the requisite performance, violating (A1).*

Consequently, we find that **no existing hardware priority queue design is suitable for modern switches** due to a fundamental limitation on either scaling (*e.g.*, PIFO), performance (*e.g.*, PIEO), or their ability to provide logical partitioning (*e.g.*, BMW-Tree).

2.1.2 SmartNICs in the Public Cloud

Another key driver for programmable packet schedulers in hardware are SmartNICs in the public cloud, either ASIC [26, 34, 37, 42] or FPGA-based [6, 15, 16, 20, 27, 35, 37]. Packet schedulers for such NICs ought to satisfy three requirements:

(B1) Scale to tens of thousands of flows: The packet scheduler on the SmartNIC may need to implement scheduling policies for a large number of active (concurrent) flows. This might seem surprising because NIC packet buffers are typically orders of magnitude smaller than those used in switches; however, unlike switches, modern SmartNICs may be required to make scheduling decisions for flows not just in their local TX or RX queues, but also those residing in *host memory*. This is a popular theme in the cloud setting where, in order to save valuable CPU cycles, the hypervisor dataplane (including scheduling functionality) is offloaded to the NIC [15]. The packet scheduler aboard the NIC is then responsible for deciding which backlogged flow queues in host memory to serve at any point, with the number of scheduling candidates scaling as (tenants $\times$ flows per tenant). For instance, across 1M+ VMs in Azure, VFP [14] reports 4.8K active connections *per VM* at the 99th percentile, and as high as 12K at P99.9. We expect the scalability problem to become all the more apparent given trends of increasing core counts [5] (and therefore potential for multitenancy), and as more services that traditionally used multiple physical NIC queues (*e.g.*, RDMA) become amenable to virtualization [19].

(B2) Sustain 100GbE+ line-rates: While state-of-the-art NICs have lower throughput requirements compared to switches, they still need to support line rates of 100 Gbps and beyond [32]. This is particularly relevant in the context of public clouds because network bandwidth is a commoditized resource and an important driver for many high-performance cloud-based applications [35].

(B3) Implement scheduling both across and within tenants: Finally, in the context of multi-tenant clouds, the NIC scheduler should be able to provide, at minimum, the ability to schedule traffic *across* tenants (to enforce cloud providers' policy requirements, *e.g.*, fairness or bandwidth quotas) and *within* tenants (to provide application-level priority queueing), without imposing significant resource overhead.

Recent priority queue designs (*e.g.*, PIEO [40], BMW-Tree [47]) symbolize a concerted effort towards addressing the *scalability* requirement outlined in (B1). For instance, we

can synthesize a PIEO instance with up to 64K entries, and a BMW-RPU instance with 350K entries on a state-of-the-art FPGA (§6.2). Unfortunately, *these designs do not meet both the performance (B2) and provider policy (B3) requirements.*

In the context of (B2), the problem with existing designs is that performance degrades rapidly as the number of elements increases, a fundamental tradeoff associated with comparison-based sorting. Moreover, scaling out using a mesh is not feasible because NICs are considerably smaller and more resource-poor than switches; as such, single-instance performance is the key factor in determining feasibility. For example, [47] reports that a single BMW-RPU instance can sustain 200 Mpps with 85K elements using a 28nm ASIC process; this is sufficient to drive line rate at 100 GbE (148.8 Mpps), but not at 200 GbE. The picture is even more dire for packet scheduling on FPGA-based SmartNICs [15, 20]. For instance, as we will show in §6.2, BMW-Tree achieves a packet rate of 55 Mpps for 85K elements on a state-of-the-art, Intel Stratix 10 MX FPGA, 37% of line rate even at 100 Gbps.

A second, more fundamental problem with these designs is that it is impossible to disentangle their *function* (implementing priority queueing) from their *form* (a fixed tree [10, 25, 47] of queue elements). As a result, implementing n distinct priority queues (*e.g.*, for n tenants) requires duplicating the underlying data structure, imposing significant resource overhead, fragmentation of queue memory, or both. As before, the key enabler for (B3) is *logical partitioning*.

Once again, we find that **existing hardware priority queues are not viable alternatives for packet scheduling in modern SmartNICs** because they do not provide the necessary performance or logical queueing functionality.

2.2 Exploring a Different Tradeoff

In this work, we seek to explore a different tradeoff that allows us to circumvent the performance-scalability barrier: sacrificing a small amount of *precision* to achieve the best of both worlds. In this regard, we are motivated by prior work's observation that a large fraction of networked applications *do not* require particularly high precision [2, 39]. For instance, both VLAN and DSCP support up to 8 traffic classes (3-bit priority tags), priority-qdisc (*tc-prio*) in the Linux kernel provides at most 16 priority bands, and state-of-the-art commercial switches support up to 32 strict-priority queues [2]. These, in turn, provide sufficiently fine-grained priority queueing to support a variety of higher-level abstractions: transport protocols and frameworks (*e.g.*, Homa [30], PASE [31], PIAS [8]), congestion and interference controllers (RC3 [29], QJUMP [18]), and high-performance overlay networks (*e.g.*, GRIN [1], SLIM [49]). In what follows, we describe the design of a highly scalable and performant priority queue exploring this tradeoff.

3 BBQ Overview

BBQ is a new hardware-based priority queue architecture for packet scheduling that is designed with three goals in mind: (1) *scalability*, the maximum number of concurrent flows that the queue can support, (2) *performance*, the maximum steady-state packet rate that the queue can sustain, and (3) *logical partitioning*, the ability to multiplex several logical queues atop a single physical queue. In this section, we give an overview of BBQ's design before diving into the architectural details in §4. We start with a brief introduction to the data structure underlying BBQ in §3.1, followed by a high-level description of the BBQ primitive in §3.2. Finally, in §3.3, we describe the challenges the hardware architecture must address in order to meet our system goals.

3.1 Data Structure

Integer Priority Queueing: BBQ leverages an *Integer Priority Queue* (IPQ) scheme to alleviate the tension between scalability and performance (§2). Unlike traditional priority queues where elements can have arbitrary priorities, an IPQ requires elements to map to a *finite set of integer priorities*, called its *priority span*; for an IPQ that supports P integer priorities, the span is represented by the set $\{0, ..., P-1\}$. Quantizing the priority range allows IPQs to implement a simple counting sort-like algorithm: the IPQ maintains an array of P *priority buckets*, each representing a unique priority in its span; when a new element is enqueued, it is inserted into the bucket indexed by the element's priority.

The only remaining challenge is to find the right priority bucket to dequeue from. Since only a subset of buckets may be occupied at any time, dequeueing entails finding the *highest-priority bucket containing at least one element*. A naive approach is to sequentially check buckets in decreasing order of priority, stopping at the first non-empty bucket; however, this may turn out to be expensive, necessitating $O(P)$ sub-operations in the worst case.

Building efficient IPQs using Find-First Set: One approach to improve the run-time efficiency of dequeue operations is to encode the occupancy of the IPQ's priority buckets as a P-bit wide bitmap, with '0's representing empty buckets, and '1's representing buckets containing at least one element. Then, the *most-significant set bit* (MSSb) in the bitmap yields the required bucket to dequeue from. Finding the MSSb, an operation known as *Find First Set* (FFS), can be performed with $\Theta(\log P)$ simple bit operations (bit-shifts and additions) using a binary search algorithm. Unfortunately, scaling to large values of P (*e.g.*, 32K) using FFS Queues quickly becomes impractical due to constraints on the maximum word size that the hardware can efficiently operate upon. For example, general-purpose processors provide FFS intrinsics for at most 64-bit words; similarly, implementing FFS on bitmaps larger than 64 bits would result in low operating frequency even in more specialized circuits (*e.g.* ASICs or FPGAs).

Scaling to larger priority spans using Hierarchical FFS:

[45] novelly observed a recursive structure to the problem: given an array of *bitmaps* ordered by priority, the *highest-priority non-zero bitmap* can also be identified via a single FFS operation using a schema similar to the one described above. This observation naturally leads to the notion of a *Hierarchical* FFS (HFFS) Queue.

The idea is to construct a tree of bitmaps, with leaf nodes representing regular FFS Queues. The bitmaps are encoded such that, at any level in the tree, *a '1' in any bit position indicates a non-empty priority bucket in the subtree rooted at that node*. Now, to dequeue the highest-priority element, we recursively perform FFS at each level of the tree starting with the root, following the MSSb until we arrive at the required priority bucket. BBQ uses a variant of the HFFS Queue as its underlying data-structure.

3.2 The BBQ Primitive

At the heart of BBQ is a priority index structure inspired by HFFS Queues: a perfect w-ary tree of w-bit bitmaps representing the queue occupancy. The bitmap tree in a BBQ can be composed of an arbitrary number of levels, which in turn dictates the queue's priority span. In general, for a BBQ with w-bit bitmaps and $D \geq 1$ levels, the number of supported priorities is $P = w^D$. Figure 2 depicts a BBQ with $w = 3$ and $D = 2$, representing $P = 3^2 = 9$ unique priorities.

The bitmap tree has a recursive structure: at the leaf level of the tree (*e.g.*, the L_2 bitmaps in Figure 2), each bit maps to a unique *priority bucket*; at non-leaf levels (*e.g.*, L_1), each bit maps to a unique subtree of bitmaps. In either case, we maintain the invariant that a bit in any bitmap is 1 *if and only if* there is a priority bucket containing at least one element in the corresponding subtree. In BBQ, we additionally associate with every bit a *subtree occupancy counter* (StOC) that indicates the total number of elements in that subtree; a StOC is non-zero if the corresponding bit is 1, and vice versa. As we will see in §4, the design choice of storing additional counters enables us to achieve stall-free execution of the BBQ pipeline, yielding high performance (i.e., full pipelining) with a relatively small memory overhead.

IPQs group queue elements (QEs) with identical priority in the same priority bucket (PB). In BBQ, PBs are implemented as *doubly-linked lists* of QEs. In particular, each PB stores a pair of pointers: `HEAD` and `TAIL`, pointing to the first and last QE in the linked list, respectively. QEs themselves are composed of two attributes: (1) a `DATA` field to store arbitrary, user-supplied identifiers[3], and (2) a pair of pointers (`PREV` and `NEXT`) to other QEs, allowing them to interface with the PBs' doubly-linked lists.

The final component of the BBQ is the *Free List* (FL): a FIFO queue containing pointers to QEs that are currently unallocated (i.e., *not* enqueued in the BBQ). Table 1 depicts the operations supported by BBQ.

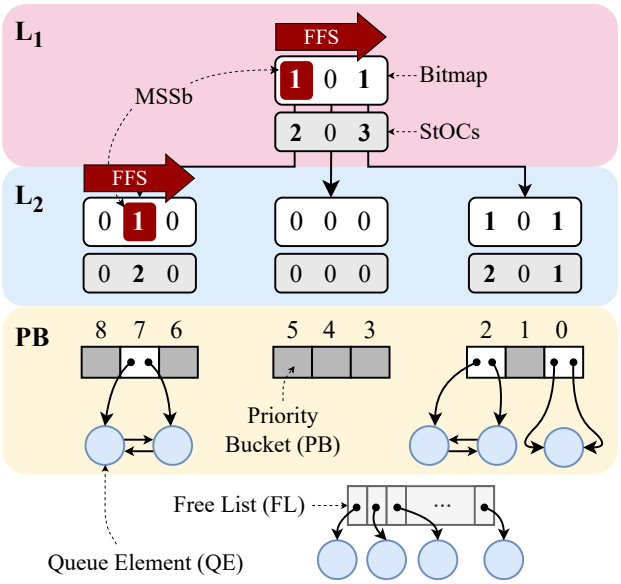


Figure 2: 2-level BBQ with $w = 3$ bit bitmaps. To dequeue the highest-priority element, we recursively perform FFS at each level of the tree starting with the root, following the most-significant set bit (MSSb) until we arrive at the required priority bucket.

3.3 Goals and Challenges

Recall that we sought out to build BBQ with three key goals in mind: *scalability*, *performance*, and *logical partitioning*. In this section, we describe how BBQ meets these goals, and the challenges the underlying hardware architecture must address to achieve them.

3.3.1 Scalability

Using an IPQ-based design breaks the dependence between run-time complexity of operations and queue size, allowing BBQ to support a large number of QEs without necessitating a fundamental performance trade-off. In many ways, scalability "falls out" of this high-level design choice, allowing us to explicitly optimize for the other goals.

3.3.2 Performance

While the data structure underlying BBQ is conceptually simple, realizing this functionality in hardware in a manner that achieves high performance is a challenging proposition.

The overall performance of the hardware queue (measured in packets per second) is the product of two independent metrics: (C_1) f_{max} or the maximum frequency that the queue operates at, which is dictated by its *critical path* (i.e., the worst-case combinational delay in the hardware circuit), and (C_2) the number of operations that can be issued every cycle. Given the logical complexity of the queue operations (i.e., `ENQUEUE` or `DEQUEUE`), it is impractical to perform them in a single hardware clock cycle because it would significantly

[3] BBQ, like most priority queues, is agnostic of the data contained in the QEs. The `DATA` attribute has a configurable bit-width and could be used to store a pointer to a packet, flow ID, or even a reference to another BBQ. Unless specified otherwise, we will assume that QEs represent *flows* [41].

Operation	Description
`ENQUEUE(X, p)`	Inserts the given data, `X`, into the BBQ with priority p.
`DEQUEUE(t)` → `(Y, p)` or `?` `t` ∈ {`MAX`,`MIN`}	Extracts the data, `Y`, corresponding to either the *highest-priority* QE (when `t` is `MAX`) or the *lowest-priority* QE (when `t` is `MIN`) currently enqueued in the BBQ. Returns `?` if empty.

Table 1: Priority Queue operations supported by BBQ.

throttle f_{max}, hurting (C_1). Instead, operations must be divided into a sequence of n *stages*, where each stage involves a smaller quantum of work. While this improves f_{max} of the resulting circuit by reducing combinational delay, doing so naively (*e.g.*, trying to avoid concurrency problems by issuing one operation every n cycles) would slash (C_2) by a factor of n, once again degrading performance.

The key to high performance – and simultaneously the most challenging aspect of BBQ's architectural design – is *pipelined parallelism*. In this context, pipelining refers to designing the hardware architecture such that *multiple stages* may simultaneously be active at the same time, thereby allowing operations to be issued fewer than n cycles apart. Unfortunately, there are several sources of complexity that make it non-trivial to achieve a high degree of pipelined parallelism: practical limitations on the number of R/W ports on physical memory blocks, data hazards (i.e., ephemeral memory dependencies between active pipeline stages), and control hazards (i.e., logical and algorithmic dependencies between stages).

Our key finding in this context is that by carefully architecting BBQ's hardware pipeline, we can, in fact, achieve fully-pipelined execution (i.e., guaranteed throughput of 1 operation per cycle) *without* compromising on the maximum clock frequency. This is realized by: (a) employing deep pipelining to preserve f_{max}, and (b) using a variety of architectural techniques (speculation, write-forwarding, and instruction coloring) to handle pipeline hazards without stalling or discarding operations. We describe the BBQ pipeline in detail in §4.

3.3.3 Logical Partitioning

Full decoupling between BBQ's queue memory (i.e., its QEs) and its priority index structure (i.e., the bitmap tree) gives BBQ a unique opportunity to provide logical partitioning with no resource overhead. The idea is to *treat the bitmap tree as a collection of n disjoint subtrees, each of which maps to an independent BBQ*. This effectively partitions the original BBQ's priority span into n disjoint regions; then, in order to `DEQUEUE` an element from a *logical* BBQ, we simply "mask" the appropriate bits in the bitmap tree while performing FFS such that we only traverse down the corresponding subtree.

This technique allows us to fully reuse *all* of the physical BBQ's resources without any performance degradation or fragmentation of queue memory. There is, however, a cost in terms of precision, because each logical BBQ can only address $\frac{1}{n}$'th of the priority span of the underlying instance. For use-cases where the number of logical partitions is not too large (*e.g.*, 32-64 port switches [33], or cloud servers hosting 16-128 tenants), this is simply a matter of appropriately provisioning the priority span of the underlying BBQ.

Since logical partitioning is, (a) a key enabler for building efficient priority queue meshes for line-rate switches and realizing hierarchical scheduling in multi-tenant cloud NICs (§7), and (b) adds negligible overhead in the BBQ datapath, we natively support this feature in the BBQ primitive. We describe logical partitioning (both for prior work, as well as BBQ) in more detail in Appendix A. Logical partitioning also enables us to extend the BBQ primitive to operate over dynamic priority ranges with zero overhead, an idea we describe in Appendix D.

4 BBQ Architecture

In this section, we describe the architectural details that enable us to map BBQ's design to hardware in a manner that achieves our performance goals. We begin by describing the hardware pipeline in §4.1, followed by a description of the hurdles that arise while trying to fully pipeline the design.

4.1 Hardware Pipeline

In principle, enqueueing and dequeueing follow a similar blueprint, yielding an intuitive algorithm for mapping them to a unified datapath: for each level of the tree starting with the root (i.e., L_1), compute the bitmap index (for `DEQUEUE(t)`, the index is computed by performing FFS on the bitmap, while for `ENQUEUE(X, p)`, it is computed using simple bit manipulations on p), increment/decrement the corresponding StOC, then update the bitmap; finally, enqueue into or dequeue from the doubly-linked list corresponding to the target priority bucket. To maximize performance, we take a careful two-pronged approach.

(1) Maximizing f_{max}: Our first objective is to maximize f_{max}, or the maximum clock frequency at which the BBQ circuit can operate. To do this, we use a *deep pipeline* where individual stages are designed to do both *little* and *roughly equal* amounts of work. Table 2 depicts the events that occur at cycle-level in a 11-stage pipeline for a 2-level BBQ.[4] By load balancing expensive operations across stages, we minimize the number and severity of same-stage dependencies (depicted by ↳ and ⇝). For instance, chaining FFS and StOC updates (multi-bit addition or subtraction) would result in a large combinational delay, so we split this work across stages

[4] Since the L_1 level has a small memory footprint, we choose to store the associated metadata (bitmaps and StOCs) in registers with single-cycle access latency. The larger arrays (*e.g.*, L_2 bitmaps and StOCs, priority buckets, and queue elements) use substantially more memory and involve more complex address decoding logic; as such, we store these in SRAM with a 2-cycle access latency in order to optimize f_{max}.

Cycle	Description	PHR
0	Register inputs If ENQUEUE: $F \leftarrow$ FreeList.POP() // Pop free list	
1	Compute L_1 bitmap index ↳ Read the corresponding L_1 StOC	L_1
2	Compute, Write: L_1 StOC ⇝ L_1 bitmap Read L_2 bitmap	L_2
3	// Read delay for L_2 bitmap	
4	Compute L_2 bitmap index ↳ Read the corresponding L_2 StOC	
5	// Read delay for L_2 StOC	
6	Compute, Write: L_2 StOC ⇝ L_2 bitmap Read the corresponding PB	PB
7	// Read delay for PB	
8	If DEQUEUE: (a) Read $X \leftarrow$ QE_{DATA}[PB.TAIL^{new}] (b) Read $Y \leftarrow$ QE_{PREV}[PB.TAIL^{new}]	
9	// Read delay for QE_{DATA} and QE_{PREV}	
10	If ENQUEUE: // Enqueue at the HEAD (a) $QE_{DATA}[F] \leftarrow$ Data to enqueue (b) Write $QE_{NEXT}[F] \leftarrow$ PB.HEAD (c) Write QE_{PREV}[PB.HEAD] $\leftarrow F$ (d) Write PB.HEAD$^{new} \leftarrow F$ If DEQUEUE: // Dequeue from TAIL (a) FreeList.PUSH(PB.TAIL) (b) Write PB.TAIL$^{new} \leftarrow Y$ (c) Output X	

Table 2: 11-stage hardware pipeline for a 2-level BBQ (without operation coloring) highlighting independent pipeline hazard regions (PHRs). ↳ and ⇝ indicate same-stage dependencies, which result in more complex combinational logic. In general, a BBQ with $D > 1$ levels entails a pipeline depth of $p = 7 + 4 \times (D-1)$ stages.

(*e.g.*, cycles 1 and 2).

(2) Maximizing operations per cycle: As described in §3.3.2, high f_{max} is only useful if we are not rate-limited by the pipeline latency or even a portion of it. Our second objective is to *fully pipeline* the BBQ design so it can concurrently process as many operations as there are pipeline stages, thereby achieving its maximum rate of 1 op/cycle. There are several challenges we encounter in this process, which we address in detail next.

4.2 A Fully-Pipelined Architecture

Pipeline Hazard Regions: The first key enabler for BBQ's stall-free architecture is the design choice of associating every bit in the bitmap tree with a *subtree occupancy counter*. Recall that for a given bit, the associated StOC indicates the *total number of elements contained in the corresponding subtree*.

To understand how this enables pipelining, consider a strawman design with n pipeline stages where we *only* store the bitmaps for each level of the HFFS tree, but not the associated StOCs. Here, the earliest time we know whether a DEQUEUE operation causes a priority bucket to become empty is when the operation is committed (i.e., the final pipeline stage). Now, consider what happens if this causes a bit in any bitmap along the path to that priority bucket to flip (i.e., become '0'). Any subsequent DEQUEUE operations in the pipeline may have been routed along the tree based on stale state, creating an incorrigible *control hazard*. As a result, we would have to either: (a) discard and re-issue the subsequent DEQUEUE operation(s), hurting worst-case performance, or (b) only issue DEQUEUEs every n cycles, defeating the purpose of pipelining altogether.

Instead, StOCs allow us to divide the BBQ pipeline into independent *pipeline hazard regions* (PHRs) mapping to different levels of the bitmap tree, as shown in Table 2. When exiting a PHR, the outcome of every operation (either an ENQUEUE or a DEQUEUE) is committed to the StOC. This has two implications: (1) two operations can only be conflicted (i.e., have data or control dependencies between them) if they are in a PHR at the same time, and (2) conflicts are limited to intra-PHR state (*e.g.*, the bitmap or StOC data at that tree level). As a result, we only have to address intra-PHR hazards (i.e., dependencies between active operations in the same PHR), which are far more localized – and therefore more tractable – than hazards spanning the entire pipeline. We characterize our implementation of StOCs in detail in Appendix B.

While StOCs *enable* us to achieve stall-free operation, they are not sufficient to guarantee a fully-pipelined design on their own. In what follows, we describe three types of intra-PHR hazards we encountered in our endeavor to fully-pipeline BBQ, and the architectural techniques used to address them.

(H_1) Data Hazards: The simplest form of hazards we encounter are *data* hazards, where one stage of the pipeline either: (a) issues a memory read, or (b) waits on completion of a memory read at an address that is concurrently updated by a different pipeline stage.

For instance, consider the BBQ pipeline depicted in Table 2. If Stage 2 issues a read for an L_2 bitmap that is simultaneously being modified by Stage 6, it will receive either a stale or invalid value 2 cycles later.[5] Similarly, if Stage 3 has a read in progress for the same memory address, it will receive a stale value on the next cycle. This is a common problem in processor design, where the standard solution – and the one we use here – is to perform ***write forwarding*** from a later pipeline stage to its predecessors when a read-after-write conflict occurs.

[5] Certain hardware platforms may guarantee a consistent memory view, but this is not true in general (*e.g.*, FPGA SRAM).

In order to resolve data hazards, we need to first compute *whether* and *which pairs of* operations in a PHR access the same state (*e.g.*, bitmaps, StOCs, PBs). BBQ exploits the hierarchical nature of the queue to make this computation efficient: since bitmap and StOC addresses also have a hierarchical structure to them (*e.g.*, the address of an L_3 bitmap is generated by splicing together the address of its L_2 parent and its own index in the parent bitmap), we can both reduce address comparator logic and improve f_{max} by memoizing address conflicts at higher levels and propagating them down the pipeline; then, when we need to compute address conflicts for lower levels, we reuse the memoized results, necessitating comparison of only the lower address bits.

(H_2) Non-atomic bitmap accesses: A second type of intra-PHR hazard arises due to the non-atomic nature of bitmap accesses, causing back-to-back `DEQUEUE` operations to be routed down incorrect paths in the bitmap tree.

To illustrate this problem, consider the example depicted in Figure 3. Initially, at ①, both the MSb and LSb of an L_2 bitmap are set, and the corresponding StOC values are 1 and 2, respectively. Now, consider two `DEQUEUE-MAX` operations, OP_A and OP_B, issued one cycle apart. Since the highest-priority (left-hand) subtree has just one element and becomes empty after the first `DEQUEUE` operation (i.e., OP_A), we would *expect* to see OP_B to be routed down the right-hand subtree (corresponding to the LSb). Instead, we find that *both* operations are incorrectly routed down the left-hand subtree. The problem arises at ③, the moment OP_B and OP_A reach Cycles 4 and 5 of the pipeline, respectively. At this point, OP_A is waiting on the read for the MSb's StOC (issued one cycle earlier, at ②) to complete, while OP_B computes the same MSSb as OP_A and issues a read for the same StOC. It is only on the following cycle – at ④, when OP_A decrements the MSb's StOC down to 0 – that we discover that the left-hand subtree becomes empty, and that OP_B *should* have been steered to the right-hand subtree instead; unfortunately, it is far too late by this point. Note that this is not simply a rare performance issue that can be addressed by, *e.g.*, discarding trailing operations in case of conflicts; rather, it is a *correctness* bug. In this case, since OP_B may have already been "committed" to StOCs earlier in the pipeline, the operation cannot simply be discarded. Once again, we would have to either: (a) stall the pipeline, hurting worst-case performance, or (b) enforce that `DEQUEUE` operations are issued at least 2 cycles apart, throttling the queue's `DEQUEUE` throughput.

To address this problem, we adopt another technique from the architecture literature: ***speculation***. The idea is as follows: within the L_i PHRs, if we have two back-to-back `DEQUEUE-MAX` operations (say, OP_A and OP_B, issued in that order, respectively) that access the same bitmap, *we compute the bitmap index for OP_B speculating that OP_A causes the MSSb to become '0'*. Consequently, OP_B will issue a read for the next-MSSb. On the following cycle, if we find that OP_A indeed caused the MSSb to flip (i.e., the corresponding StOC

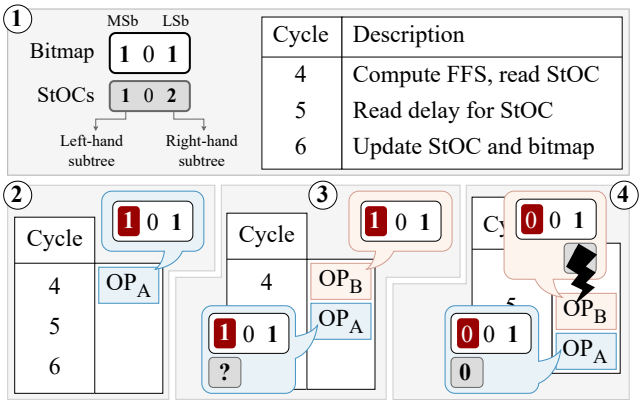


Figure 3: Non-atomic read-modify-write accesses to bitmaps cause OP_B (the second of two consecutive `DEQUEUE-MAX` operations) to be incorrectly routed to the left-hand subtree.

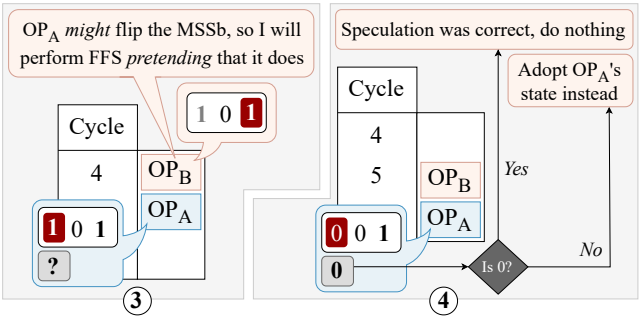


Figure 4: If there are conflicting operations in the L_i PHRs, operations issued later compute bit indices *speculating* that earlier operations will change the bitmap.

becomes 0), our speculation was correct, and OP_B proceeds as usual. Otherwise, OP_B simply discards its own state, and adopts both the MSSb index and StOC values computed by OP_A for the remainder of the pipeline. The speculation logic is illustrated in Figure 4.

The key observation here is that instead of squandering OP_B's one available read on data that are going to be available anyway (via inter-stage forwarding), we can "hedge" our bet on multiple bits simultaneously, ensuring that at least one of them yields the desired outcome. We also note that, while the description presented here involves only `DEQUEUE-MAX` operations for the sake of simplicity, the same technique generalizes to any combination and order of operations (i.e., `ENQUEUE`, `DEQUEUE-MIN`, and `DEQUEUE-MAX`).

(H_3) Non-atomic PB accesses: Conceptually similar to (H_2), the third and final type of hazard arises due to the non-atomic nature of priority bucket accesses, causing back-to-back `DEQUEUE` operations to potentially corrupt state within the *PB* PHR. To see why, consider stages $6-10$ of the BBQ pipeline. In Stage 10, `DEQUEUE` operations cause the PB's `TAIL` pointer to be updated (now denoted by PB.`TAIL`$^{\text{new}}$). In stage 8, `DEQUEUE` operations perform a read that is *supposed to be addressed* by the most up-to-date `TAIL` pointer for the

corresponding PB. However, consider what happens when two back-to-back DEQUEUE operations (say, OP_A and OP_B, issued in that order, respectively) land at the same priority bucket. At Stage 9, OP_B is waiting on completion of the read addressed by PB.TAIL available on the previous cycle. However, OP_A, now at Stage 10, modifies the TAIL pointer, causing the read issued by OP_B (for QE_{DATA} and QE_{PREV}) to become stale. Unfortunately, write-forwarding does not help here because the stale variable (PB.TAIL) is being used to address other state, creating a control hazard.

To address this problem, we introduce the notion of ***operation coloring*** (inspired by *graph coloring*, problems where vertices in a graph must be assigned colors such that no two adjacent vertices have the same color), which works as follows. First, we tag each operation with a *Color* attribute, which assumes one of two values: *Purple* (■) or *Orange* (●). An operation's color determines which end of the PB's doubly-ended linked-list it interacts with: operations colored purple operate on the HEAD of the PB, while those colored orange operate on the TAIL. All ENQUEUE operations are always colored purple, while DEQUEUE operations are, by default, colored orange. Finally, we add a single constraint on color: *a DEQUEUE operation must not have the same color as a conflicting operation issued immediately before it (i.e., one cycle earlier)*. In the first cycle of the PB PHR (Cycle 7), if the active operation is a DEQUEUE that conflicts with another operation in the subsequent pipeline stage, it is recolored. Table 3 depicts the relevant portion of the BBQ pipeline post operation coloring (extraneous details are omitted for the sake of brevity).

C	Description	
7	// Color operation	
8	If DEQUEUE ■: Rd QE_{DATA}[PB.HEAD^{new}] Rd QE_{NEXT}[PB.HEAD^{new}]	If DEQUEUE ●: Rd QE_{DATA}[PB.TAIL^{new}] Rd QE_{PREV}[PB.TAIL^{new}]
9	// Read delay	
10	If ENQUEUE ■: Update PB.HEAD ^{new}	
	If DEQUEUE ■: Update PB.HEAD ^{new}	If DEQUEUE ●: Update PB.TAIL ^{new}

Table 3: Updated stages 7 – 10 showing operation coloring.

Observe that, so long as the DEQUEUE operations are colored differently from any operation *immediately preceding them in the pipeline*, they will not incur stale reads. The key idea here is that each operation (whether an ENQUEUE or a DEQUEUE) only affects one pointer in the PB's (HEAD, TAIL) pair.[6] As a result, picking a mutually exclusive color also guarantees exclusivity on the data structure itself. Operations spaced more than one cycle apart can be safely handled via write forwarding. An artifact of this design choice is that back-to-back DEQUEUEs landing at the same priority bucket will not dequeue elements in FIFO order; however, since DEQUEUEs are bound to be interleaved with ENQUEUEs during typical operation, we do not expect this case to arise often.

Together, these optimizations realize a fully-pipelined priority queue architecture with both high f_{max}, and a guaranteed operation throughput of 1 op/cycle independent of workload.

[6]The only situations in which *both* pointers are affected is when the priority bucket becomes empty (i.e., due to a DEQUEUE), or a priority bucket that was *previously* empty becomes non-empty (i.e., due to an ENQUEUE). Speculation precludes the first possibility (attempting to DEQUEUE an empty PB), so we are left with the second corner case, which we handle explicitly.

5 $BBQ_{\odot}$: A Latency-Free BBQ

While deep pipelining is key to BBQ's high performance, the resulting *pipeline latency* (i.e., the number of clock cycles that elapse between when an operation is issued and when it completes) introduces a new source of error in the relative ordering of elements compared to an "ideal" priority queue.

The problem manifests due to a confluence of two factors: (a) since it takes several cycles for an operation to traverse the pipeline, in order to use BBQ at full throughput (1 op/cycle), multiple operations need to be issued concurrently; and (b) for a pipeline of depth p, *the minimum delay* for a high-priority element to be dequeued, served, and re-enqueued into the queue is also p. *Consequently, any DEQUEUE operations that are issued in the p cycle interval that the highest-priority element is not present in the queue might ultimately dequeue lower-priority elements*. A concrete example of the problem is described in §C.1. The aforementioned problem is inextricably tied to our decision of using a pipelined architecture, implying that the BBQ primitive alone cannot guarantee absolute accuracy at full throughput.

However, we find that a simple augmentation to the BBQ primitive allows us to hide this latency and avoid the accuracy issues that come with it: use a tiny PIFO as a "cache" in front of the BBQ to hold the highest-priority elements. Whenever this tiny PIFO overflows, it "leaks" the lowest priority element to the BBQ. This PIFO only needs to be able to hold as many elements as the BBQ's pipeline depth (order of tens of elements). Because of its small size, this instance does not face the scalability limitations associated with the PIFO architecture and only adds a small footprint to the design.

The augmented design, $BBQ_{\odot}$, *provably guarantees zero loss in accuracy (i.e., any dequeued element is always the highest-priority one in the system at that time) while providing full throughput (1 op/cycle)*. We can show this by proving the sufficient condition in Theorem 1: with a PIFO whose size exceeds the pipeline latency of the BBQ, the highest-priority element is always served from the PIFO, such that we never experience the accuracy or latency artifacts introduced by the accompanying BBQ. We describe $BBQ_{\odot}$ in detail in §C.2.

Theorem 1 (Priority Set Invariant for $BBQ_{\odot}$). *In a $BBQ_{\odot}$ instance composed of a BBQ with pipeline latency p cycles and a PIFO of size $k > p$, the top $(k-p)$ highest-priority elements are always in the PIFO.*

The proof can be found in §C.3.

6 Evaluation

We now evaluate BBQ. Our main goal is to understand BBQ's *performance* and *viability* for both ASICs and FPGA designs. We also compare BBQ with PIFO [41], PIEO [40], and BMW-Tree [47]. PIFO is the state-of-the-art hardware priority queue in terms of *throughput* while BMW-Tree is the state of the art in terms of *scalability*. Throughout this evaluation, we show that BBQ can surpass PIFO's throughput while achieving similar scalability to BMW-Tree.

6.1 Setup and Methodology

We implement BBQ in SystemVerilog. Given the recurring and composable structure of the design, we implement a Python script to automatically generate different configurations of BBQs by stitching together modular blocks of hand-written SystemVerilog code. Users can specify the number of levels in the tree (D), the bitwidth of every node (W), and the maximum number of elements (N). We synthesize BBQ targeting both an FPGA (Intel Stratix 10 MX FPGA [24]) as well as an ASIC. The Stratix 10 MX contains 702,720 Adaptive Logic Modules (ALMs), 140 Mb of SRAM, and two 100 Gb Ethernet ports. For comparison, we also synthesize PIFO, PIEO, and BMW-Tree targeting the same board. For each design and configuration, we conduct a bisection search to find the maximum clock frequency achievable with 3 MHz precision, picking the best synthesis across 10 seeds. To synthesize the FPGA we use Intel Quartus [21]. To synthesize the ASIC, we use Synopsys Design Compiler [43] using a 7 nm Standard Cell Library [44] based on the ASAP7 PDK [13].

All the designs we evaluate have deterministic performance that is independent of the workload. As such, our analysis focuses on the packet rate that each design is able to sustain, as well as the cost [36] (in terms of die area and FPGA resources).

6.2 FPGA

As mentioned in §2.1.2, FPGA-based NICs are increasingly used as a way to achieve programmable offloads on the NIC [15, 16, 23, 35, 37]. Having a programmable packet scheduler on the NIC would allow administrators to change the packet scheduling algorithms at run time. In this section, we evaluate how BBQ and the baseline designs perform, in terms of throughput (§6.2.1) and FPGA resources (§6.2.2). We also explore how the different BBQ design parameters affect its performance when running on an FPGA (§6.2.3).

6.2.1 Throughput Scalability

Queue capacity can influence throughput by increasing the hardware critical path, which in turn reduces the maximum clock frequency that we can achieve with the design (f_{max}). To understand this effect, we synthesize BBQ (with 8-bit bitmaps) and the baselines while changing both the queue capacity and the number of bits used to express the priorities (precision). We report both the clock frequency as well as the overall throughput achievable by each design.

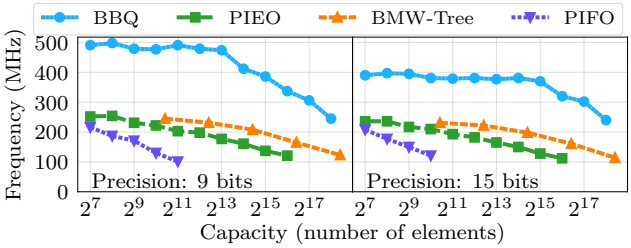


Figure 5: Clock frequency as we scale the queue capacity.

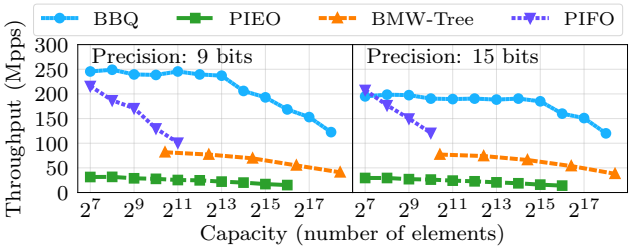


Figure 6: Throughput as we scale the queue capacity.

Figure 5 shows the clock achievable by each design when we increase the queue capacity. BBQ achieves a clock as high as 500 MHz with 9-bit precision and 400 MHz with 15-bit precision, significantly higher than the baseline designs. Moreover, BBQ is able to scale to up to 2^{17} elements while still sustaining a 300 MHz clock. In comparison, PIFO can only scale to up to 2^{11} elements.

In addition to achieving a higher clock frequency, BBQ's fully pipelined design allows it to execute an operation (enqueue or dequeue) every clock cycle. As a result, the throughput difference is even higher compared to BMW-Tree (that consumes 1 cycle to enqueue and 2 cycle to dequeue) and PIEO (that consumes 4 cycles to enqueue and 4 cycles to dequeue). However, different from the other designs, PIFO is able to execute both an enqueue and a dequeue operation in the same cycle, allowing its packet rate to match its clock frequency. Figure 6 shows the throughput of the different designs for different queue capacities. BBQ is able to drive 100 Gbps line rate (148.8 Mpps) with as many as 2^{17} elements. Also note that PIFO is able to achieve similar throughput to BBQ, but only for small queues (128 elements or less).

6.2.2 Resource Scalability

We also evaluate how the different designs scale in terms of FPGA resources. We report both ALM utilization and SRAM blocks, both as a fraction of the overall number of resources available in the target FPGA. Figure 7 shows how the ALM utilization scales as we increase the number of elements that the queue can support. In BBQ, scaling the queue capacity has little effect on the logic utilization. This is a direct consequence of BBQ's use of an integer priority queue, which lets it avoid comparison-based sorting. BMW-Tree's

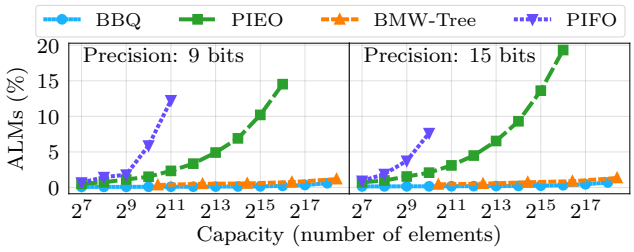


Figure 7: ALM utilization as we scale the queue capacity.

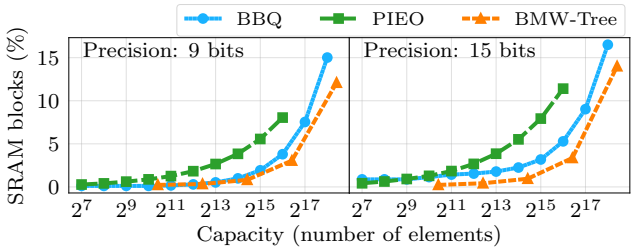


Figure 8: SRAM block utilization as we scale the queue capacity. PIFO is not included in the plot as it does not use SRAM.

hierarchical design also gives it much better scalability, using only slightly more ALM resources than BBQ. In contrast, both PIFO and PIEO use significantly more resources as we scale the capacity.

While BBQ's ALM utilization remains low even for 131,072 elements (2^{17}), BBQ, like PIEO and BMW-Tree, relies on SRAM to store its elements. As a result, we expect SRAM utilization to increase with the queue capacity for all these designs. Figure 8 shows this effect. Note that BBQ's SRAM utilization is in between PIEO's and BMW-Tree's. However, PIEO and BMW-Tree require multiple copies in order to match BBQ throughput, which causes them to use vastly more SRAM if provisioned to meet the same performance target.

6.2.3 BBQ Sensitivity Analysis

To understand the impact of BBQ's configuration on performance, we perform a sensitivity analysis of the bitmap tree parameters: the number of levels (D), and the bitmap width (W) (recall that the number of priorities is computed as $P = W^D$), while keeping the number of elements fixed.

Figure 9 depicts how BBQ's throughput behaves as a function of the bitmap width. We sweep the number of levels in the bitmap tree from $D = 3$ to 15 (or until we reach 2^{15} priorities) for different bitmap widths. Then we plot the attained throughput for the corresponding priority count. We find that bitmap widths between 2 and 8 yield similar performance (with 4 being optimal), but this deteriorates as the bitmap width increases. In particular, starting with $W = 16$, *FFS computation* becomes the primary f_{max} bottleneck. We can similarly infer from the same graph that level count has little impact on performance; for instance, we observe that a $(D = 4, W = 2)$ BBQ achieves the same throughput as a $(D = 8, W = 2)$ BBQ despite the latter containing 16× as

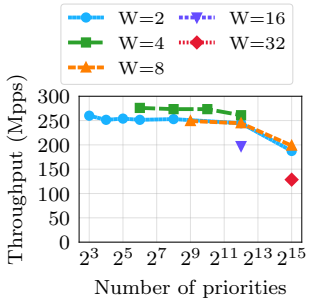


Figure 9: BBQ throughput as we increase the number of priorities when using different bitmap widths (W).

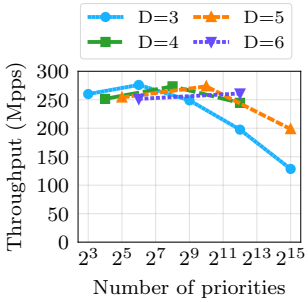


Figure 10: BBQ throughput as we increase the number of priorities when using different number of levels (D).

many priorities. Figure 10 shows the complementary view of the data, depicting change in throughput as a function of priorities for different numbers of tree levels.

6.3 ASIC

We now evaluate BBQ in the context of designs targeting ASICs. Here we compare BBQ with PIFO for two reasons: (1) it is one of the two baseline designs that supports logical partitioning, which, as we discussed in §2, is essential to allow them to be efficiently incorporated into state-of-the-art switches, and (2) it offers the best throughput among all the baseline designs.

We synthesize both BBQ and PIFO using a 7 nm process. PIFO only meets timing at 1 GHz with up to 2^{11} elements, which is consistent with [41]. BBQ meets timing at 3.1 GHz with 2^{17} elements, but we did not try scaling beyond this point. The difference in the clock frequency achieved by BBQ and PIFO means that BBQ is able to run at 55% higher throughput. To evaluate the cost of the design, we compare the chip area when synthesizing a single queue. We use the synthesis results to calculate the area used by the logic gates and estimate the SRAM area using the cost of 0.027mm^2/Mb reported by TSMC for their 7 nm process [12, 46].

Table 4 shows the chip area breakdown, divided in logic and SRAM for both BBQ and PIFO, using 9b and 15b priorities. BBQ uses very little area with logic; most of its area is taken up by SRAM. BBQ is not only able to scale to many more elements than PIFO, but also consumes less area when both are provisioned for the same capacity.

Design	Elements	Priorities	Clock	Area (mm^2)		
				Logic	SRAM	Total
PIFO	2^{11}	2^9	1 GHz	0.043		0.043
	2^{11}	2^{15}	1 GHz	0.058		0.058
BBQ	2^{11}	2^9	3.1 GHz	0.00071	0.0029	0.0037
	2^{11}	2^{15}	3.1 GHz	0.0011	0.035	0.036
	2^{17}	2^9	3.1 GHz	0.00095	0.24	0.24
	2^{17}	2^{15}	3.1 GHz	0.0014	0.29	0.29

Table 4: Chip area for the different designs. BBQ uses little logic, causing its area to be primarily determined by SRAM.

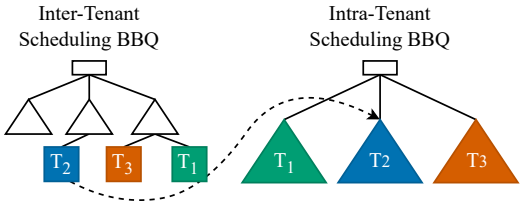


Figure 11: A two-level hierarchical NIC scheduler using BBQ. The first BBQ schedules traffic *across* tenants. The second BBQ (which is logically partitioned) is used to schedule traffic *within* each tenant.

7 Applications

In Section 2, we motivated the need for a hardware priority queue capable of supporting packet scheduling for two rapidly emerging use-cases: terabit-scale switches, and NICs in multi-tenant cloud datacenters. Having evaluated its scalability, throughput, and resource usage, in this section we describe how BBQ can fill these application-level gaps.

7.1 Packet Scheduling on Switches

As described in §2.1.1, a key enabler for priority queue deployment in modern switches is the ability to realize *logical partitioning*, allowing the packet scheduler to leverage the simpler priority queue mesh architecture depicted in Figure 1(b). BBQ achieves this by treating disjoint subtrees in its priority index structure as independent queues (§3.3.3), enabling multiplexing of the underlying instance; as noted earlier, this comes at a cost in terms of precision (each logical queue gets $\frac{1}{k}$'th the original priority range), but with no resource overhead, performance penalty, or fragmentation of queue memory.

In terms of *performance* and *scalability*, we can synthesize BBQ instances with 100K+ entries and 32K priorities at 3.1GHz using a 7nm ASIC process; at two operations per packet, each BBQ instance can sustain a packet rate of 1.55Bpps. Our target switch (NVIDIA SN4700, 400GbE x32) supports an aggregated packet rate of 8.4Bpps, implying that a total of $\lceil \frac{8.4}{1.5} \rceil \times 2 = 12$ BBQs that are logically partitioned among the 32 output ports in a shared scheduler pipeline are sufficient to match the switch fabric's processing speed. Based on our ASIC synthesis results for a single BBQ (§6.3), we estimate that provisioning each of these 12 instances with 131K queue entries and 32K priorities (1K priorities per port) would require a total area of 3.48mm^2. Considering that a switch chip area ranges from 200mm^2 to 800mm^2 [41] this corresponds to 0.4–1.74% of the total chip area.

Thus, BBQ's scalability, performance, and ability to be logically partitioned make it, for the first time, a plausible candidate to realize priority queueing in modern switches.

7.2 Packet Scheduling on Cloud SmartNICs

A key requirement for NIC-based packet schedulers in public clouds is the ability to independently perform scheduling both *across* tenants and *within* each tenant (§2.1.2). By allowing several logical BBQs (each representing one tenant) to share a single BBQ instance, we can efficiently use the available resources (*e.g.*, queue memory) without having to provision one priority queue per tenant. In order to enforce cross-tenant traffic policies, we instantiate another, smaller BBQ that stores references to the tenant BBQs. Conceptually, this corresponds to the two-level hierarchical scheduler depicted in Figure 11, with the lower and upper levels handling intra- and inter-tenant scheduling decisions, respectively.

Previously, we evaluated the feasibility of operating BBQ on an ASIC in the context of switches (§7.1). We now frame our discussion about scalability and performance for SmartNICs in the context of the more resource-constrained device family: FPGAs. On an Intel Stratix 10MX we can synthesize a BBQ instance with 100K+ entries and 32K priorities that meets timing at 302MHz, and uses 0.45% of the available ALMs and and 9% of the total FPGA SRAM, respectively. Consequently, a single instance can sustain 151Mpps, surpassing 100GbE line rate with minimum-sized packets (148.8 Mpps).

On FPGAs, scaling to higher line rates (*e.g.*, 400GbE [22]) is beyond the capability of any single priority queue instance, and would require augmenting the scheduler pipeline with multiple BBQs. However, given the resource cost of each instance relative to total FPGA resources, this is currently impractical; any priority queue design would have to significantly reduce its SRAM footprint (*e.g.*, offloading elements to DRAM) in order to make scaling out on FPGAs practical.

8 Related Work

Counting priority index: The data structure used in BBQ is similar to the counting priority index (CPI) proposed by Wang and Lin [45]. They were the first to hypothesize that an integer priority queue could be used to speed up packet scheduling in both software and hardware. Unfortunately, CPI is not implementable in hardware in its original form as it fails to account for the many practical issues that arise when building a pipelined hardware design, e.g., memory access latency, hazards, and limited memory. As we discussed in §3 and §4, the challenging aspects of BBQ's design stem from these very issues. BBQ is also orthogonal to Eiffel [38], which deals with the practical issues of using a priority index to schedule packets in *software*.

Hardware priority queueing: PIFO [41] is the current state-of-the-art priority queue implementation in terms of throughput, and BMW-RPU [47] is the current state-of-the-art in terms of scalability. As confirmed in our evaluation, BBQ is able to match PIFO's throughput while scaling beyond BMW-RPU's maximum capacity. Another notable hardware priority queue design is pHeap [10], unfortunately pHeap can only process an operation every two clock cycles, which makes it unsuitable for line-rate switches. Further, besides PIFO and PIEO, none of the designs that close the gap in terms of scalability (including BMW-Tree and pHeap) support logical partitioning efficiently, making them impractical for

deployment in both switches and SmartNICs in the cloud setting. We characterize the efficiency that existing designs achieve in implementing logical partitioning in §A.1.

Approximate priority queueing: There is also a line of work that proposes approximating different scheduling algorithms to make them amenable to hardware implementation [2, 3, 17, 39, 48]. BBQ borrows from these works the observation that a small priority set (at the hardware level) is sufficient for most use cases. These works provide interesting theoretical insights and a path to implement some scheduling policies on existing programmable switches. However, BBQ's design shows that it is unnecessary to sacrifice accuracy in order to achieve scalability and speed.

9 Discussion

Limitations: A key limitation of IPQ-based designs such as BBQ is that they operate over priority ranges that are both *finite*[7] and *static*. While we can, in fact, augment the vanilla BBQ primitive to support *dynamic* priority ranges (Appendix D), boundedness of the priority span remains an immutable constraint. The fundamental reason is that we must map every priority in BBQ to a bucket in physical memory, so SRAM usage scales linearly with the priority span. Notably, this scheme becomes altogether impractical when the required precision grows beyond a certain threshold (*e.g.*, supporting 2^{32} priorities would require over 500MB of SRAM *just* to store bitmaps). In §6, we demonstrated the feasibility of synthesizing a BBQ instance with 15-bit priority tags (32K priorities), but we don't expect this number to scale much further. Thus, the ideal operating point for BBQ corresponds to a setting where we need to support a *large number of queue entries falling in a small (possibly dynamic) priority range*. Some priority queue architectures also enable richer abstractions (*e.g.*, PIEO's predicate-based filtering allows scheduling based on eligibility criteria such as virtual or wall-clock time [40]), which BBQ does not support.

Future Work: Today, we are at an inflection point with regard to the scalability of hardware priority queue designs. On the one hand, support for 100K+ queue entries is the culmination of a decade-long concerted effort towards jointly optimizing scalability and performance. On the other hand, this appears to be the end of the scalability roadmap: since every queue element must be stored *somewhere*, we are ultimately bottlenecked by available memory. For the sake of performance, today's designs exclusively use SRAM, which offers deterministic, single-cycle memory access. However, SRAM is a scarce resource, and even highly scalable designs such as BBQ and BMW-Tree [47] would require over 10% of the available FPGA SRAM (§6.2) to support 200K queue entries – a highly impractical proposition. However, given the trend of increasing multi-tenancy in datacenters, it is not far fetched to believe that schedulers will some day need priority queueing for 1M+ flows. A natural question then is: *how do we get there?* We believe the key to this lies in offloading queue entries to DRAM, which provides much slower (and non-deterministic) access latencies compared to SRAM, but is a far more abundant memory resource. The clean decoupling between BBQ's priority index structure and its queue memory (i.e., BBQ's ability to locate the highest-priority entry without needing access to the entry itself) makes it feasible to offload queue memory to DRAM, but there are several challenges that need to be addressed along the way. We leave it to future work to realize this lofty goal.

[7] Technically, priority ranges are *always* finite (regardless of the underlying priority queue design) because they are ultimately upper-bounded by the maximum precision afforded by the priority tag (i.e., number of priority bits). However, the point here is that comparison-based priority queue designs (*e.g.*, PIFO) can, in principle, create the *illusion* of an infinite priority range using large priority tags; for instance, a time-based PIFO scheduler that uses nanosecond-granularity timestamps as priorities would require well over 500 years to exhaust a 64-bit priority range ($2^{64} = 1.8 \times 10^{19}$ priorities). Conversely, IPQs hit their priority scaling limits far earlier than any reasonable interpretation of infinity.

10 Conclusion

PIFO's vision—a programmable packet scheduler that operates at line rate even on high-throughput switches—has been hampered by throughput and scalability issues of existing priority queue designs. In this paper, we presented BBQ, a new priority queue design that is both scalable and fast. At the heart of its design is an integer priority queue that allows BBQ to avoid the complexity barrier imposed by comparison-based sorting. While this paper shows the usefulness of BBQ for performing packet scheduling in both FPGA SmartNICs and line rate switch ASICs, we expect such a high-performance priority queue to find use in many other contexts. We look forward to future work that creatively use BBQ for other purposes.

Acknowledgements

We thank our shepherd, Mina Arashloo, and the anonymous reviewers for their insightful feedback. We are also indebted to Anup Agarwal for inspiring us to write this paper, Andrew Boutros for his guidance on ASIC design, Weina Wang for her help with the theory, Vyas Sekar for shaping the story, and the Azure Host Networking and Scheduler teams (particularly Mahmoud Elhaddad, Idan Regev, and Dongwook Lee) for valuable discussions on design and use-cases. This work was funded by Intel and VMware through the Intel/VMware Crossroads 3D-FPGA Academic Research Center, a VMWare Systems Research Award, NSF Award 2007733, a Cylab Presidential Fellowship, and a Google Research Gift.

References

[1] Alexandru Agache, Razvan Deaconescu, and Costin Raiciu. Increasing datacenter network utilisation with GRIN. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 29–42, 2015.

[2] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. SP-PIFO: Approximating Push-In First-Out behaviors using Strict-Priority queues. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 59–76, Santa Clara, CA, February 2020. USENIX Association.

[3] Albert Gran Alcoz, Balázs Vass, Gábor Rétvári, and Laurent Vanbever. Everything matters in programmable packet scheduling. *arXiv preprint arXiv:2308.00797*, 2023.

[4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 435–446, New York, NY, USA, 2013. Association for Computing Machinery.

[5] AMD. AMD EPYC 4th gen 9004 & 8004 series server processors – details, 2023. https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series.html#specs.

[6] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 93–109, Santa Clara, CA, February 2020. USENIX Association.

[7] Nirav Atre, Hugo Sadok, Erica Chiang, Weina Wang, and Justine Sherry. SurgeProtector: Mitigating temporal algorithmic complexity attacks using adversarial scheduling. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 723–738, New York, NY, USA, August 2022. Association for Computing Machinery.

[8] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Weicheng Sun. Pias: Practical information-agnostic flow scheduling for data center networks. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, page 1–7, New York, NY, USA, 2014. Association for Computing Machinery.

[9] J. C. R. Bennett and Hui Zhang. WF2Q: Worst-case fair weighted fair queueing. In *Proceedings of IEEE INFOCOM '96. Conference on Computer Communications*, volume 1 of *INFOCOM '96*, pages 120–128 vol.1, 1996.

[10] R. Bhagwan and B. Lin. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings IEEE INFOCOM 2000 Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Society*, volume 2 of *INFOCOM 2000*, pages 538–547 vol.2, 2000.

[11] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. Association for Computing Machinery.

[12] Jonathan Chang, Yen-Huei Chen, Wei-Min Chan, Sahil Preet Singh, Hank Cheng, Hidehiro Fujiwara, Jih-Yu Lin, Kao-Cheng Lin, John Hung, Robin Lee, Hung-Jen Liao, Jhon-Jhy Liaw, Quincy Li, Chih-Yung Lin, Mu-Chi Chiang, and Shien-Yang Wu. A 7nm 256Mb SRAM in high-k metal-gate FinFET technology with write-assist circuitry for low-VMIN applications. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 206–207, 2017.

[13] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal*, 53:105–115, 2016.

[14] Daniel Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, pages 315–328, Boston, MA, March 2017. USENIX Association.

[15] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pages 51–66, Renton, WA, April 2018. USENIX Association.

[16] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. Corundum: An open-source 100-Gbps NIC. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '20, pages 38–46. IEEE, 2020.

[17] Peixuan Gao, Anthony Dalleggio, Yang Xu, and H. Jonathan Chao. Gearbox: A hierarchical packet scheduler for approximate weighted fair queuing. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '22, pages 551–565, Renton, WA, April 2022. USENIX Association.

[18] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 1–14, 2015.

[19] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. MasQ: RDMA for virtual private cloud. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pages 1–14, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Intel. Intel, Baidu drive intelligent infrastructure transformation, 2020. https://www.intel.com/content/www/us/en/newsroom/news/baidu-intelligent-infrastructure-transformation.html#gs.5vl4ru.

[21] Intel. FPGA Design Software – Intel Quartus Prime, 2023. https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html.

[22] Intel. Intel Agilex 7 FPGAs and SoCs product brief, 2023. https://www.intel.com/content/www/us/en/content-details/762901/intel-agilex-7-fpgas-and-socs-product-brief.html.

[23] Intel. Intel infrastructure processing unit (Intel IPU) platform (codename: Oak Springs Canyon), 2023. https://www.intel.com/content/www/us/en/products/platforms/details/oak-springs-canyon.html.

[24] Intel. Intel Stratix 10 MX 2100 FPGA, 2023. https://ark.intel.com/content/www/us/en/ark/products/210297/intel-stratix-10-mx-2100-fpga.html.

[25] Aggelos Ioannou and Manolis G. H. Katevenis. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Transactions on Networking*, 15(2):450–461, April 2007.

[26] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 243–259. USENIX Association, November 2020.

[27] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From luna to solar: The evolutions of the compute-to-storage networks in Alibaba Cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 753–766, New York, NY, USA, 2022. Association for Computing Machinery.

[28] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Universal packet scheduling. In *13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '16, pages 501–521, 2016.

[29] Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Recursively cautious congestion control. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 373–385, Seattle, WA, April 2014. USENIX Association.

[30] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

[31] Ali Munir, Ghufran Baig, Syed Mohammad Irteza, Ihsan Ayyub Qazi, Alex X Liu, and Fahad Rafique Dogar. Pase: synthesizing existing transport strategies for near-optimal data center transport. *IEEE/ACM Transactions on Networking*, 25(1):320–334, 2016.

[32] Nvidia. ConnectX-7 400G Adapters: Smart, accelerated networking for modern data center infrastructures, 2023. https://nvdam.widen.net/s/csf8rmnqwl/infiniband-ethernet-datasheet-connectx-7-ds-nv-us-2544471.

[33] Nvidia. Nvidia spectrum sn4000 series switches, 2023. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/br-sn4000-series.pdf.

[34] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. SENIC: Scalable NIC for end-host rate limiting. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, pages 475–488, Seattle, WA, April 2014. USENIX Association.

[35] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. Ensō: A streaming interface for NIC-application communication. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '23, pages 1005–1025, Boston, MA, July 2023. USENIX Association.

[36] Hugo Sadok, Aurojit Panda, and Justine Sherry. Of apples and oranges: Fair comparisons in heterogenous systems evaluation. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, pages 1–8, New York, NY, USA, 2023. Association for Computing Machinery.

[37] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. We need kernel interposition over the network dataplane. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 152–158, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. Eiffel: Efficient and flexible software packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 17–32, Boston, MA, February 2019. USENIX Association.

[39] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 685–699, Santa Clara, CA, February 2020. USENIX Association.

[40] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 367–379, New York, NY, USA, 2019. Association for Computing Machinery.

[41] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 44–57, New York, NY, USA, 2016. Association for Computing Machinery.

[42] Brent Stephens, Aditya Akella, and Michael M. Swift. Loom: Flexible and efficient NIC packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 33–46, Boston, MA, February 2019. USENIX Association.

[43] Synopsys. Design Compiler, 2023. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[44] Vinay Vashishtha, Manoj Vangala, and Lawrence T. Clark. ASAP7 predictive design kit development and cell design technology co-optimization: Invited paper. In *2017 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD, pages 992–998, 2017.

[45] Hao Wang and Bill Lin. Per-flow queue management with succinct priority indexing structures for high speed packet scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 24(7):1380–1389, 2013.

[46] Shien-Yang Wu, C.Y. Lin, M.C. Chiang, J.J. Liaw, J.Y. Cheng, S.H. Yang, C.H. Tsai, P.N. Chen, T. Miyashita, C.H. Chang, V.S. Chang, K.H. Pan, J.H. Chen, Y.S. Mor, K.T. Lai, C.S. Liang, H.F. Chen, S.Y. Chang, C.J. Lin, C.H. Hsieh, R.F. Tsui, C.H. Yao, C.C. Chen, R. Chen, C.H. Lee, H.J. Lin, C.W. Chang, K.W. Chen, M.H. Tsai, K.S. Chen, Y. Ku, and S. M. Jang. A 7nm CMOS platform technology featuring 4th generation FinFET transistors with a 0.027um2 high density 6-T SRAM cell for mobile SoC applications. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 2.6.1–2.6.4, 2016.

[47] Ruyi Yao, Zhiyu Zhang, Gaojian Fang, Peixuan Gao, Sen Liu, Yibo Fan, Yang Xu, and H. Jonathan Chao. BMW tree: Large-scale, high-throughput and modular PIFO implementation using balanced multi-way sorting tree. In *Proceedings of the ACM SIGCOMM 2023 Conference*, SIGCOMM '23, pages 208–219, New York, NY, USA, 2023. Association for Computing Machinery.

[48] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 179–193, New York, NY, USA, August 2021. Association for Computing Machinery.

[49] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS kernel support for a low-overhead container overlay network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 331–344, 2019.

Appendix A Logical Partitioning in Practice

In §2, we motivated *logical partitioning* (i.e., the ability to multiplex several logical priority queues atop a single physical queue) as a key requirement for deployability in modern switches and SmartNICs. In this section, we first characterize the extent to which existing priority queue designs can realize logical partitioning (§A.1), followed by a detailed description of the mechanisms that enable BBQ to achieve this functionality (§A.2).

A.1 Existing Designs

The goal of logical partitioning is to allow a single, physical priority queue to emulate a collection of multiple, *logically-independent* priority queues. Simply realizing this abstraction is not particularly challenging, but doing so *efficiently* turns out to be a major impediment for most priority queue designs. We can evaluate efficiency along three axes: (1) *queue fragmentation*, or the worst-case fraction of queue elements (QEs) lost to external fragmentation when an instance is partitioned q ways; (2) *performance overhead*, or the throughput degradation resulting from logical partitioning; and, (3) *resource overhead*, or the resource cost (*e.g.*, logic, memory) required to support q logical partitions relative to an unpartitioned priority queue.

PIFO supports logical partitioning with zero queue fragmentation, a small performance overhead, and a resource overhead that scales linearly with queue size: Consider the example depicted in Figure 12, where a physical PIFO (provisioned with $N = 8$ QEs) is partitioned into $q = 2$ logical PIFOs. Per usual, PIFO maintains a sorted list of QEs ordered by priority [41]. To realize logical partitioning, PIFO annotates every QE with a *Logical PIFO ID* (in this case, q_0 or q_1) at enqueue time. Then, in order to dequeue from the i'th logical PIFO, it first "selects" the subset of elements with the corresponding ID (q_1 in our example), then performs priority decoding to extract the highest-priority element from that subset. Since every QE in the physical instance can always be independently addressed by every logical PIFO, queue memory is fully multiplexed, yielding zero fragmentation. Every element must be annotated with a $\log_2 q$ bit wide ID, resulting in a resource overhead that scales with queue size. Finally, selecting the appropriate subset of elements involves an extra comparator per element, incurring a small performance cost.

With modest changes, PIEO can support logical partitioning with zero queue fragmentation, and performance/resource overheads that scale with queue size: Architecturally, PIEO is organized as a matrix: an array of $2\sqrt{N}$ *sublists*, each consisting of $\sqrt{N}$ QEs sorted by priority. Besides standard priority queue operations, PIEO allows specifying *eligibility predicates*: a programmable function that "selects" a subset of elements to dequeue from. In principle, this is similar to the mechanism PIFO uses to implement logical partitioning (Figure 12), and an appropriate predicate

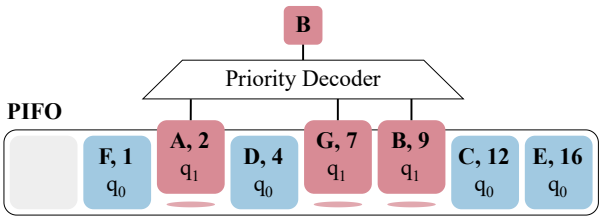


Figure 12: A single physical PIFO partitioned into 2 logical PIFOs: q_0 consisting of 4 elements, and q_1 consisting of 3 elements. Available queue memory is fully multiplexed among the logical PIFOs, resulting in zero fragmentation.

function can be used in PIEO to the same effect. Unfortunately, the *vanilla PIEO design does not allow QEs belonging to different logical PIEOs to coexist in the same sublist*. The reason is that, as a first step, PIEO must perform predicate filtering at sublist level, which only supports range-based queries (*e.g.*, $a \leq f \leq b$) but not set queries (*e.g.*, $f \in X$) that are required for logical partitioning. Consequently, it would incur external fragmentation at the granularity of sublists. However, we note that this is not a fundamental limitation, and with minor changes to the design (*e.g.*, by annotating every sublist with a q-bit bitmap representing the logical PIEO QEs contained therein), PIEO can, in theory, achieve logical partitioning with no external fragmentation while incurring a resource/performance cost similar to PIFO.

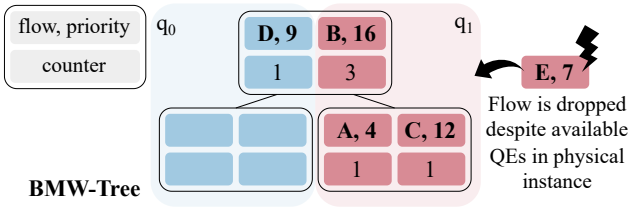


Figure 13: A 2-level 2-way BMW-Tree instance partitioned into 2 logical queues. Each logical queue must be mapped to a *physical* subtree, resulting in external fragmentation. Once a subtree becomes full, enqueues into the corresponding logical queue are impossible even if there are available QEs in other subtrees.

Other comparison-based priority queue designs cannot efficiently implement logical partitioning: PIFO and PIEO both satisfy a property that is key to achieving zero fragmentation in fixed-layout priority queues: maintaining a total ordering over elements at all times. *In their effort to leverage spatial and pipelined parallelism, other designs violate this property, inducing significant queue fragmentation*. We illustrate this point using the 2-way BMW-Tree depicted in Figure 13. To implement partitioning between $q = 2$ logical BMW-Trees, each logical queue must be statically mapped to a *physical* subtree as shown in the figure;[8] any other arrangement might result in inadvertently dequeueing from the wrong logical queue. The result is that, for a BMW-Tree instance of size N, each logical queue can address only $\frac{N}{q}$

[8] Heap property is intentionally violated at root level to enable partitioning.

QEs, incurring a fragmentation cost that scales with q. For instance, with $q = 8$ partitions, we would risk losing up to 87% of the queue memory to external fragmentation (and still not be able to support N flows). Instead, over-provisioning the physical instance to account for the worst case (all N elements being enqueued in a single logical queue, while the other 7 queues remain empty) would incur a 700% memory overhead. Other tree-like priority queue designs, such as pHeap [10] and Pipelined Heap [25], encounter precisely the same issue.

Overall, we find that besides the designs that maintain a total ordering over elements (PIFO and PIEO), *realizing logical partitioning in comparison-based designs incurs prohibitively high resource overhead, queue fragmentation cost, or both.*

A.2 Logical Partitioning in BBQ

Given a BBQ instance with w-bit bitmaps, we illustrate how to partition it into q logical BBQs by means of two exemplar configurations: one where $q \leq w$, and another where $q > w$.

(1) Fewer logical partitions than the bitmap width ($q \leq w$): Consider a 2-level BBQ with 4-bit bitmaps (i.e., $w = 4$, $D = 2$) that we would like to partition into $q = 2$ logical BBQs. The physical BBQ has a priority span of $P = 4^2 = 16$ priorities. As a first step, we partition this range equally between the two logical instances, allocating priorities $[0, 7]$ to queue q_0, and $[8, 15]$ to q_1. Observe that, in order to facilitate this split, the L_1 bitmap also needs to be partitioned as shown in Figure 14, with the lower two bits corresponding to q_0, and the upper two bits corresponding to q_1. Enqueueing an entry, X, into logical queue $i \in [0, 1]$ with relative priority $j \in [0, 7]$ is simple: first, we compute the *absolute* priority corresponding to the physical BBQ as $p = (i \times 8) + j$,[9] then perform `ENQUEUE(X, p)` as usual. In order to dequeue the highest (or lowest) priority entry from the i'th logical queue, we first mask the bits in the root bitmap *not* corresponding to q_i (*e.g.*, for $i = 1$, we would apply the mask `0xC`), perform FFS on them, then proceed down the bitmap tree as in a typical `DEQUEUE` operation.

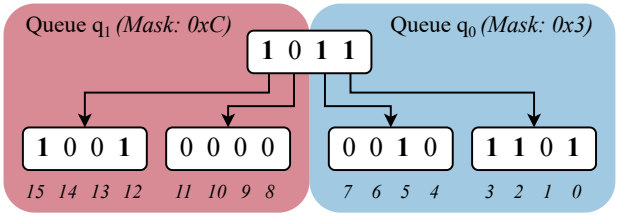


Figure 14: Bitmap tree for a 2-level BBQ with 4-bit bitmaps partitioned into 2 logical BBQs, q_0 and q_1. Each logical BBQ is allocated disjoint ranges of 8 priorities. To `DEQUEUE` from a logical BBQ, we first apply the corresponding *mask* to the L_1 bitmap before performing FFS on it.

(2) More logical partitions than the bitmap width ($q > w$): Consider again a ($w = 4$, $D = 2$) BBQ that we would now like to partition into $q = 8$ logical BBQs. Observe that each bit in the root bitmap now corresponds to *two* different logical BBQs, and therefore does not offer any discriminatory power.[10] Consequently, we eliminate this level of the bitmap tree; in its place, we insert a single pipeline stage that steers operations to their respective subtrees based on the logical queue index (*e.g.*, `ENQUEUE` and `DEQUEUE` operations on $i \in \{2, 3\}$ are steered to the second-from-right subtree). The updated bitmap tree structure is depicted in Figure 15. Each L_2 bitmap now maps to $q' = 2$ different logical BBQs, and we apply the idea described in (1) (since $q' \leq w$) to achieve this partitioning.

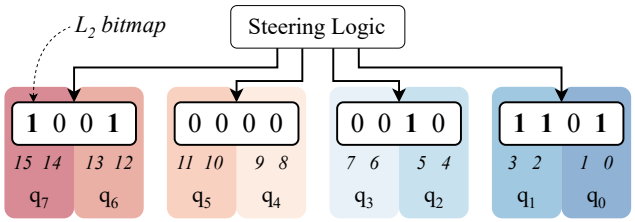


Figure 15: Bitmap tree for a 2-level BBQ with 4-bit bitmaps partitioned into 8 logical BBQs. The root (L_1) bitmap no longer adds any value, so we replace it with a *steering stage* that simply routes operations on logical BBQs to the corresponding subtree.

Thus, with nominal changes to the BBQ pipeline, we can support a broad range of partitioning configurations without any performance overhead. In contrast to PIFO and PIEO, the resource cost (corresponding to over-provisioning the priority range) scales with the *degree of logical partitioning* rather than queue size. Finally, full decoupling between its priority index structure (i.e., the bitmap tree) and QEs enables BBQ to achieve zero queue fragmentation. Overall, these techniques make it possible for BBQ to *efficiently* realize logical partitioning.

Appendix B Using StOCs in Practice

In §4.2, we described how every bit in BBQ's bitmap tree is associated with a StOC, which represents the *total number of elements contained in the corresponding subtree*. StOCs are an important component in BBQ because they are a key enabler for its fully-pipelined architecture. In this section, we characterize two practical details regarding our implementation of StOCs: how they are sized (§B.1), and a general optimization technique that improves their performance (§B.2).

B.1 Sizing

In order to handle the worst case (i.e., all elements in the BBQ being contained in a single priority bucket), every StOC must be provisioned to represent the range $[0, N]$, where N is the number of supported queue elements. Conventionally, N is configured to be a power of two (i.e., 2^k) to avoid wasting resources such as pointer address bits. However, in the case

[9] Logically, this corresponds to simply concatenating together i and j.

[10] Since either of the logical BBQs contained therein may be empty, a '1' in any bit position of the L_1 bitmap provides no guarantee that a `DEQUEUE` operation on that subtree will succeed.

of BBQ, naively provisioning the queue with $N = 2^k$ elements would entail $(k+1)$-bit StOCs, with the most-significant bit (MSb) only ever being used to encode the maximum occupancy of 2^k. Instead, in BBQ, we snap N to a value of the form $(2^k - 1)$, allowing us to use k-bit StOCs. Thus, carefully sizing the queue (and deliberately wasting one element's worth of address space) saves 1 bit per StOC, yielding a sizeable reduction in memory footprint.

B.2 Waterlevel Bit Optimization

Since StOC bit-widths scale with the queue size (§B.1), performing arithmetic or logical operations on these counters can be expensive for large BBQ instances. As we will see, these operations sometimes need to be chained together with other combinational logic in a single pipeline stage, which in turn inflates the critical path and significantly degrades f_{max}. In this section, we describe a general optimization technique that alleviates counter-related performance bottlenecks, yielding up to 17% higher f_{max} for some BBQ configurations.

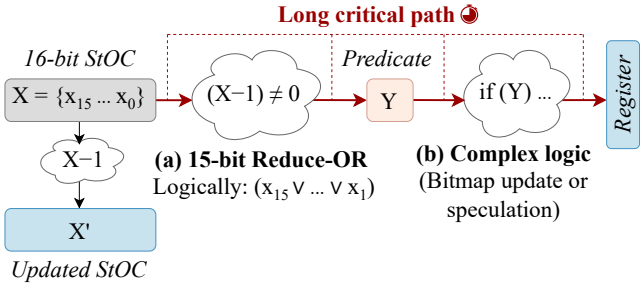


Figure 16: Dependency chain involving 16-bit counter logic for a `DEQUEUE` operation. The critical path comprises of (a) a 15-bit Reduce-OR (to determine whether the StOC becomes 0), chained with (b) more combinational logic (which uses (a) as a predicate).

To illustrate the problem, consider a `DEQUEUE` operation in stage 6 of the BBQ pipeline depicted in Table 2. This stage comprises of several sub-operations, two of which we will focus on here: (a) decrementing an L_2 StOC and checking if the resulting value becomes zero, and (b) updating the L_2 bitmap predicated on the result of (a). Note that (a) operates on a $log_2(N+1)$-bit counter and (b) operates on a w-bit bitmap, and chaining these sub-operations together inevitably puts them on the critical path. The problem is further exacerbated if (b) entails more complex combinational logic (*e.g.*, resolving speculation outcomes, which, as shown in step ④ of Figure 4, follows the same blueprint). The critical path for this sequence of sub-operations is depicted in Figure 16.

To address this performance bottleneck, we augment every StOC in BBQ with an additional bit called the **waterlevel bit** (WLb). We maintain the invariant that the WLb is set to '1' if the current StOC value is greater than or equal to 2, otherwise it is set to '0'. The key idea is that *the WLb opportunistically memoizes the future result of (a)*,[11] allowing

[11] Observe that, for positive values of X, the expression evaluated by (a), $(X-1) \neq 0$, is logically equivalent to $X \geq 2$, the value encoded in the WLb.

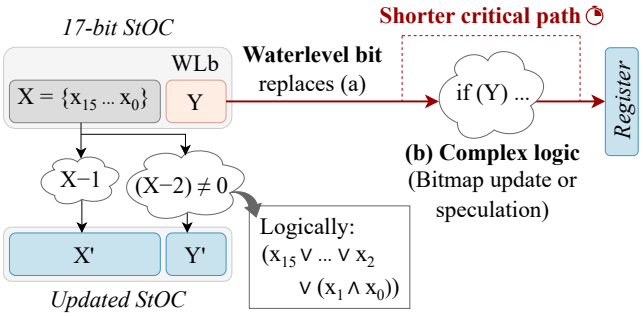


Figure 17: The waterlevel bit (WLb) replaces (a), improving f_{max} by removing the counter operations from the critical path.

it to directly serve as the predicate for (b) instead of having to compute it from scratch (Figure 17). This avoids chaining the expensive Reduce-OR computation with other complex combinational logic, thereby shrinking the critical path. Moreover, the updated value of the WLb (corresponding to the current StOC value minus 2) can be efficiently computed using another 16-bit operation; however, since this is not chained with additional logic, it does not appear on the critical path.

In the context of BBQ, this optimization yields between $5-17\%$ higher f_{max} on the Stratix 10 MX FPGA for configurations with $(2^{17}-1)$ queue entries (i.e., 17-bit StOCs). This does come at a resource cost, since every StOC must now be one bit wider to accommodate the WLb (corresponding to approximately 4.75kB higher SRAM usage for a BBQ that supports 32K priorities with 8-bit bitmaps). However, we find that the resulting performance improvements justify this resource overhead, and we enable this optimization by default in the BBQ artifact.

Finally, we note that the underlying technique – memoizing useful counter arithmetic results in the counter structure itself – is a general one that may benefit *any* design which employs occupancy counters that might ultimately appear on the critical path (*e.g.*, BMW-Tree [47]).

Appendix C BBQ$_\odot$: A Latency-Free BBQ

In §5, we briefly described the latency artifacts that arise due to pipelining, and how they might cause the packet schedule produced by BBQ to deviate from that of an "ideal" priority queue. In this section, we describe BBQ$_\odot$, an augmentation of BBQ that counteracts the latency issue. We start with a concrete example motivating the problem (§C.1), then dive into BBQ$_\odot$'s design (§C.2), followed by a proof of its correctness (§C.3).

C.1 Motivating Example

Consider the scenario depicted in Figure 18, where we use a BBQ instance with a pipeline latency of $p = 4$ cycles to implement strict priority scheduling at a bottleneck switch. There is a single high-priority flow, A, competing with 3 lower-priority flows (B, C, and D); if none of the flows are application-limited, we expect A to receive the full share of

bandwidth, while the other flows should starve (i.e., never be served). Assume now that a single packet can be transmitted every other cycle (i.e., line rate corresponds to one packet every two cycles). Since it takes 4 cycles for the BBQ instance to complete a DEQUEUE request and yield the appropriate flow to schedule, we are faced with two alternatives for managing priority queue state.

First, when a flow is scheduled, we might re-enqueue the flow in the BBQ and immediately issue another DEQUEUE request, wait 4 cycles for the BBQ to respond with the next flow to schedule, and so on. This guarantees accuracy (i.e., the scheduled flow is always the highest-priority one), but implies that flows can only be scheduled every 4 cycles, wasting half the link bandwidth.[12]

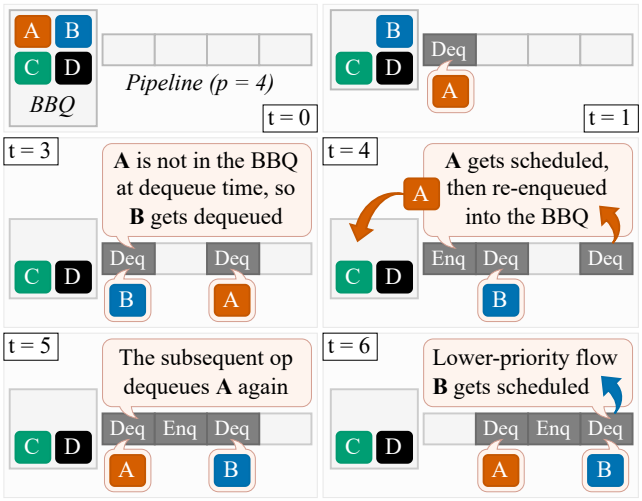


Figure 18: Issuing concurrent DEQUEUE requests, in combination with BBQ's pipeline latency, incorrectly causes a lower-priority flow, B, to be extracted (at $t = 3$) and scheduled (at $t = 6$).

The second option is to preemptively maintain *multiple* concurrent DEQUEUE requests in flight such that a flow is *always* available to be scheduled. In our example, this corresponds to issuing DEQUEUE operations 2 cycles apart such that a flow gets dequeued every other cycle (*e.g.*, at $t = 5$ in Figure 18). While this saturates the link bandwidth, it also implies that *not all active flows are enqueued in the BBQ at dequeue time*. For instance, at $t = 1$, the first issued DEQUEUE operation extracts flow A from the BBQ. Consequently, at $t = 3$, the next DEQUEUE operation results in the extraction of a lower-priority flow, B. Flow A eventually returns to the BBQ at $t = 4$, but it is far too late by this point: at $t = 6$, the second DEQUEUE operation completes, yielding B. In effect, this violates the strict priority scheduling requirement we sought to enforce.

[12] Alternatively, we might schedule *bursts* of packets at a time so as to hide the pipeline latency, but this effectively imposes a leaky bucket atop the underlying scheduling policy, which may not always be desirable.

C.2 BBQ$_{\odot}$ Design

BBQ$_{\odot}$ is composed of two components: a BBQ and a PIFO instance, which are connected as shown in Figure 19. To simplify the theoretical analysis of this system (§C.3), we assume that both components run at the same clock frequency (say 250MHz), and the system clock, denoted by CLOCK$_{sys}$ runs at half the frequency. On each CLOCK$_{sys}$ cycle, we can insert one element into the BBQ$_{\odot}$, extract the highest-priority element, or both.

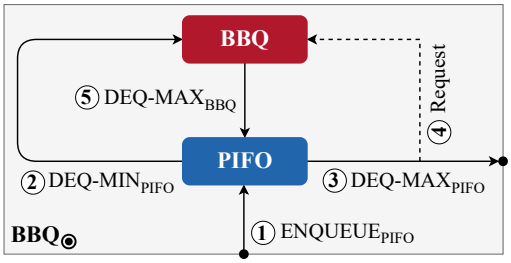


Figure 19: BBQ$_{\odot}$ design.

At a high level, our goal is to keep the PIFO full, only inserting into the BBQ when low-priority elements "spill over" from the PIFO. When a new element arrives at the system, it is first ENQUEUE'd into the PIFO ①; if this causes the PIFO to overflow its capacity (k), we perform a DEQUEUE-MIN$_{PIFO}$ operation ②, inserting the resulting element into the BBQ. Extracting the highest-priority element from the system involves two steps: (a) we perform a DEQUEUE-MAX$_{PIFO}$ operation to get the highest-priority element in the PIFO ③, which completes immediately, and (b) we issue a DEQUEUE-MAX$_{BBQ}$ request to fetch the highest-priority element from the BBQ ④, which takes p timesteps to complete (where p is the pipeline latency of the BBQ in units of CLOCK$_{sys}$ cycles). Finally, when a DEQUEUE-MAX$_{BBQ}$ operation completes ⑤, we insert the resulting element into the PIFO; as before, if this causes the PIFO to overflow, we move the lowest-priority element in the PIFO to the BBQ. Observe that on every CLOCK$_{sys}$ cycle (corresponding to 2 clock cycles for each component), we issue at most 2 BBQ operations and 4 PIFO operations (2 ENQUEUEs and 2 DEQUEUEs), which matches their respective operation throughputs.

C.3 Proof of Theorem 1

In this section, we prove a sufficient condition for BBQ$_{\odot}$ to guarantee zero accuracy loss when using an appropriately-sized PIFO. We start by proving a lower bound on PIFO occupancy when the associated BBQ is not empty, followed by an invariant regarding the subset of priorities contained in the PIFO at any time. In the remainder of the section, we use $P(t)$ to denote the set of elements contained in the PIFO at time t, and $|P(t)|$ to denote the cardinality of this set (and transitively the PIFO occupancy).

Lemma 1 (Lower-Bound on PIFO Occupancy). *In a BBQ$_{\odot}$ instance composed of a BBQ with pipeline latency*

p cycles and a PIFO of size $k > p$, if $|P(t)| < (k-p)$, the BBQ is empty at time t.

Proof. The intuition behind the Lemma is that, so long as the BBQ is *not* empty, it will prevent the PIFO occupancy from dropping below a certain threshold (corresponding to $k-p$). We prove this claim via contradiction, showing that if $|P(t)| < (k-p)$ starting with a full PIFO (a necessary condition for the BBQ to be non-empty), at least one DEQUEUE-MAX$_{\text{BBQ}}$ request resulted in ? after p timesteps, implying that the BBQ is empty at time t.

Let t_1 denote the *latest* time that the PIFO was full, and let t_2 denote the *earliest* time such that $|P(t_2)| < (k-p)$. Note that no elements are inserted in the BBQ in the period $[t_1, t_2]$; otherwise $\exists\, t_1' > t_1$ where the PIFO is still full, implying that t_1 was not the latest time. Assume towards a contradiction that a total of n_1 DEQUEUE-MAX$_{\text{PIFO}}$ operations and $n_2 \geq 0$ ENQUEUE$_{\text{PIFO}}$ operations were performed in $[t_1, t_2]$, and n_3 DEQUEUE-MAX$_{\text{BBQ}}$ operations completed in the same period, all of which returned non-? values.

$$n_1 - (n_2 + n_3) > p, \tag{1}$$
$$n_3 \geq \max(0, n_1 - p) \tag{2}$$

where (1) is true because the difference between the number of departures from the PIFO (n_1) and the number of arrivals to the PIFO ($n_2 + n_3$) in $[t_1, t_2]$ must correspond to an occupancy drop from k to $|P(t_2)| < k-p$, i.e., exceeding p. (2) is true because a DEQUEUE-MAX$_{\text{BBQ}}$ request is issued for every DEQUEUE-MAX$_{\text{PIFO}}$ operation, and the maximum number of requests still outstanding is at most p. Substituting (1) into (2), we get: $n_3 \geq (n_1 - p) > n_2 + n_3$, a contradiction. □

We are now ready to prove Theorem 1 (restated below for reference).

Theorem 1 (Priority Set Invariant for BBQ$_{\odot}$). *In a BBQ$_{\odot}$ instance composed of a BBQ with pipeline latency p cycles and a PIFO of size $k > p$, the top $(k-p)$ highest-priority elements are always in the PIFO.*

Proof. Given Lemma 1, we only need to consider the scenario where $(k-p) \leq |P(t)| \leq k$. Assume that the BBQ is not empty, otherwise all $|P(t)| \geq (k-p)$ elements in the PIFO are trivially the highest-priority ones. Now, assume towards a contradiction that only the $m < (k-p)$ highest-priority elements in the system are in the PIFO at a certain time t_2. It follows that the highest-priority element in the BBQ, x, has a higher priority than the remaining $|P(t_2)| - m$ elements in the PIFO at t_2. Observe that x may only have been inserted in the BBQ if, at the time of insertion, t_1: (1) the PIFO was full, and (2) all k elements in the PIFO had higher priority than x.

Now, for x to be the $(m+1)$'th highest-priority element in the system at time t_2, we must have performed $n = (k-m) > (k-(k-p)) = p$ number of DEQUEUE-MAX$_{\text{PIFO}}$ operations since t_1, implying that n $(> p)$ DEQUEUE-MAX$_{\text{BBQ}}$ operations were issued in the interval $[t_1, t_2]$. Since at most one DEQUEUE-MAX$_{\text{BBQ}}$ operation can be issued every timestep and each such operation takes exactly p timesteps to complete, it follows that *at least one* DEQUEUE-MAX$_{\text{BBQ}}$ operation completed and returned x. Thus, x is in the PIFO at time t_2, a contradiction. □

Appendix D Dynamic Priority Ranges

As described in §3.1, the standard BBQ primitive operates over a priority range that is both *finite* and *static*. While this is sufficient in some contexts (*e.g.*, strict priority scheduling), many policies implicitly assume an infinite priority set (*e.g.*, fair queueing). IPQs are fundamentally incapable of upholding this assumption (§9). Fortunately, prior work has shown that in most cases, a *dynamic* – albeit finite – priority range is sufficient to realize these policies [38, 39]. In this section, we describe how BBQ can be extended to provide the abstraction of a *rolling priority window*.

To handle dynamic priority ranges, we directly adapt Eiffel's [38] idea of using a *Circular Hierarchical FFS-based Queue* (cFFS). The idea is to have two independent HFFS queues, each with priority span P, working in tandem: a primary HFFS queue, q_0, that stores elements with priorities $[0, P)$, and a secondary HFFS queue, q_1, mapping to elements with priorities in $[P, 2P)$ (i.e., just outside q_1's range). Together, these queues represent a *logical* priority window of $[0, 2P)$. Once the primary queue becomes completely empty, the logical priority window advances by P, and the queue designations are swapped, with q_0 – now the secondary queue – buffering elements with priorities in $[2P, 3P)$, and so on and so forth.

We follow precisely the same blueprint for BBQ, with logical partitioning enabling us to multiplex both q_0 and q_1 atop a single physical BBQ instance with no resource overhead (or modification to the primitive, for that matter). The only additional component required is a simple controller to orchestrate the two logical queues. We do not implement this feature as part of our research artifact (yet), but we expect it to have little to no impact on BBQ's performance.