Eiffel: Efficient and Flexible Software Packet Scheduling

Ahmed Saeed[†], Yimeng Zhao[†], Nandita Dukkipati[*], Mostafa Ammar[†], Ellen Zegura[†],
Khaled Harras[‡], Amin Vahdat[*]
[†]*Georgia Institute of Technology,* [*]*Google,* [‡]*Carnegie Mellon University*

Abstract

Packet scheduling determines the ordering of packets in a queuing data structure with respect to some ranking function that is mandated by a scheduling policy. It is the core component in many recent innovations to optimize network performance and utilization. Our focus in this paper is on the design and deployment of packet scheduling in software. Software schedulers have several advantages over hardware including shorter development cycle and flexibility in functionality and deployment location. We substantially improve current software packet scheduling performance, while maintaining flexibility, by exploiting underlying features of packet ranking; namely, packet ranks are integers and, at any point in time, fall within a limited range of values. We introduce Eiffel, a novel programmable packet scheduling system. At the core of Eiffel is an integer priority queue based on the Find First Set (FFS) instruction and designed to support a wide range of policies and ranking functions efficiently. As an even more efficient alternative, we also propose a new approximate priority queue that can outperform FFS-based queues for some scenarios. To support flexibility, Eiffel introduces novel programming abstractions to express scheduling policies that cannot be captured by current, state-of-the-art scheduler programming models. We evaluate Eiffel in a variety of settings and in both kernel and userspace deployments. We show that it outperforms state of the art systems by 3-40x in terms of either number of cores utilized for network processing or number of flows given fixed processing capacity.

1 Introduction

Packet scheduling is the core component in many recent innovations to optimize network performance and utilization. Typically, packet scheduling targets network-wide objectives (e.g., meeting strict deadlines of flows [34], reducing flow completion time [14]), or provides isolation and differentiation of service (e.g., through bandwidth allocation [40, 35] or Type of Service levels [44, 15, 32]). It is also used for resource allocation within the packet processing system (e.g., fair CPU utilization in middleboxes [56, 30] and software switches [33]).

Packet scheduling determines the ordering of packets in a *queuing data structure* with respect to some *ranking function* that is mandated by a *scheduling policy*. In particular, as packets arrive at the scheduler they are *enqueued*, a process that involves ranking based on the scheduling policy and ordering the packets according to the rank. Then, periodically, packets are *dequeued* according to the packet ordering. In general, the dequeuing of a packet might, for some scheduling policies, prompt recalculation of ranks and a reordering of the remaining packets in the queue. A packet scheduler should be *efficient* by performing a minimal number of operations on packet enqueue and dequeue thus enabling the handling of packets at high rates. It should also be *flexible* by providing the necessary abstractions to implement as many scheduling policies as possible.

In modern networks, hardware and software both play an important role [23]. While hardware implementation of network functionality will always be faster than its corresponding software implementation, software schedulers have several advantages. First, the short development cycle and flexibility of software makes it an attractive replacement or precursor for hardware schedulers. Second, the number of rate limiters and queues deployed in hardware implementations typically lags behind network needs. For instance, three years ago, network needs were estimated to be in the tens of thousands of rate limiters [46] while hardware network cards offered 10-128 queues [4]. Third, software packet schedulers can be deployed in multiple platforms and locations, including middleboxes as Virtual Network Functions and end hosts (e.g., implementation based on BESS [33], or OpenVSwitch [45]). Hence, we assert that software solutions will always be needed to replace or augment hardware schedulers [19, 36, 47, 22, 39]. However, as will be discussed in Section 2, current software schedulers do not meet our efficiency and flexibility objectives.

Our focus in this paper is on the design and implementation of efficient and flexible packet scheduling in software. The need for programmable schedulers is rising as more sophisticated policies are required of networks [27, 50] with schedulers deployed at multiple points on a packet's path. It has proven difficult to achieve scheduler efficiency in software schedulers, especially handling packets at high line rates, without limiting the supported scheduling policies [47, 50, 36, 47, 19, 22]. Furthermore, CPU-efficient implementation of even the simplest scheduling policies is still an open problem for most platforms. For instance, kernel packet pacing can cost CPU utilization of up to 10% [47] and up to 12% for hierarchical weighted fair queuing scheduling in

NetIOC of VMware's hypervisor [37]. This overhead will only grow as more programmability is added to the scheduler, assuming basic building blocks remain the same (e.g., OpenQueue [39]). The inefficiency of these systems stems from relying on $O(\log n)$ comparison-based priority queues.

At a fundamental level, a scheduling policy that has m ranking functions associated with a packet (e.g., pacing rate, policy-based rate limit, weight-based share, and deadline-based ordering) typically requires m priority queues in which this packet needs to be enqueued and dequeued [49], which translates roughly to $O(m \log n)$ operations per packet for a scheduler with n packets enqueued. We show how to reduce this overhead to $O(m)$ for any scheduling policy (i.e., constant overhead per ranking function).

Our approach to providing both flexibility and efficiency in software packet schedulers is two fold. First, we observe (§2) that packet ranks can be represented as integers that at any point in time fall within a limited window of values. We exploit this property (§3.1.1) to employ integer priority queues that have $O(1)$ overhead for packet insertion and extraction. We achieve this by proposing a modification to priority queues based on the Find First Set (FFS) instruction, found in most CPUs, to support a wide range of policies and ranking functions efficiently. We also propose a new approximate priority queue that can outperform FFS-based queues for some scenarios (§3.1.2). Second, we observe (§3.2) that packet scheduling programming models (i.e., PIFO [50] and OpenQueue [39]) do not support per-flow packet scheduling nor do they support reordering of packets on a dequeue operation. We augment the PIFO scheduler programming model to capture these two abstractions.

We introduce Eiffel, an efficient and flexible software scheduler that instantiates our proposed approach. Eiffel is a software packet scheduler that can be deployed on end-hosts and software switches to implement any scheduling algorithm. To demonstrate this we implement Eiffel (§4) in: 1) the kernel as a Queuing Discipline (qdisc) and compare it to Carousel [47] and FQ/Pacing [26] and 2) the Berkeley Extensible Software Switch (BESS) [8, 33] using Eiffel-based implementations of pFabric [14] and hClock [19]. We evaluate Eiffel in both settings (§5). Eiffel outperforms Carousel by 3x and FQ/Pacing by 14x in terms of CPU overhead when deployed on Amazon EC2 machines with line rate of 20 Gbps. We also find that an Eiffel-based implementation of pFabric and hClock outperforms an implementation using comparison-based priority queues by 5x and 40x respectively in terms of maximum number of flows given fixed processing capacity and target rate.

2 Background and Objectives

In modern networks, packet scheduling can easily become the system bottleneck. This is because schedulers are burdened with the overhead of maintaining a large number of buffered packets sorted according to scheduling policies. Despite the growing capacity of modern CPUs, packet processing overhead remains a concern. Dedicating CPU power to networking takes from CPU capacity that can be dedicated to VM customers especially in cloud settings [28]. One approach to address this overhead is to optimize the scheduler for a specific scheduling policy [26, 25, 19, 47, 22]. However, with specialization two problems linger. First, in most cases inefficiencies remain because of the typical reliance on generic default priority queues in modern libraries (e.g., RB-trees in kernel and Binary Heaps in C++). Second, even if efficiency is achieved, through the use of highly efficient specialized data structures (e.g., Carousel [47] and QFQ [22]) or hybrid hardware/software systems (e.g. SENIC [46]), this efficiency is achieved at the expense of programmability. The Eiffel system we develop in this paper is designed to be both efficient and programmable. In this section we examine these two objectives, show how existing solutions fall short of achieving them and highlight our approach to successfully combine efficiency with flexibility.

Efficient Priority Queuing: Priority queuing is fundamental to computer science with a long history of theoretical results. Packet priority queues are typically developed as comparison-based priority queues [26, 19]. A well known result for such queues is that they require $O(\log n)$ steps for either insertion or extraction for a priority queue holding n elements [52]. This applies to data structures that are widely used in software packet schedulers such as RB-trees, used in kernel Queuing Disciplines, and Binary Heaps, the standard priority queue implementation in C++.

Packet queues, however, have the following characteristics that can be exploited to significantly lower the overhead of packet insertion and extraction:

- *Integer packet ranks:* Whether it is deadlines, transmission time, slack time, or priority, the calculated rank of a packet can always be represented as an integer.
- *Packet ranks have specific ranges:* At any point in time, the ranks of packets in a queue will typically fall within a limited range of values (i.e., with well known maximum and minimum values). This range is policy and load dependent and can be determined in advance by operators (e.g., transmission time where packets can be scheduled a maximum of a few seconds ahead, flow size, or known ranges of strict priority values). Ranges of priority values are diverse ranging from just eight levels [1], to 50k for a queue implementing per flow weighted fairness which requires a number of priorities corresponding to the number of flows (i.e., 50k flows on a video server [47]), and up to 1 million priorities for a time indexed priority queue [47].
- *Large numbers of packets share the same rank:* Modern line rates are in the range of 10s to 100s of Gbps. Hence, multiple packets are bound to be transmitted with nanosecond time gaps. This means that packets with small differences in their ranks can be grouped and said to have

System	Efficiency	HW/SW	Flexibility				Notes
			Unit of Scheduling	Work-Conserving	Supports Shaping	Programmable	
FQ/Pacing qdisc [26]	$O(\log n)$	SW	Flows	No	**Yes**	No	Only non-work conserving FQ
hClock [19]	$O(\log n)$	SW	Flows	**Yes**	**Yes**	No	Only HWPQ Sched.
Carousel [47]	$O(1)$	SW	Packets	No	**Yes**	No	Only non-work conserving sched.
OpenQueue [39]	$O(\log n)$	SW	Packets & Flows	**Yes**	No	On enq/deq	Inefficient building blocks
PIFO [50]	$O(1)$	HW	Packets	**Yes**	**Yes**	On enq	Max. # flows 2048
Eiffel	$O(1)$	SW	Packets & Flows	**Yes**	**Yes**	On enq/deq	-

Table 1: Proposed work in the context of the state of the art in scheduling

the same rank with minimal or no effect on the accurate implementation of the scheduling policy. For instance, consider a busy-polling-based packet pacer that can dequeue packets at fixed intervals (e.g., order of 10s of nanoseconds). In that scenario, packets with gaps smaller than 10 nanoseconds can be considered to have the same rank.

These characteristics make the design of a packet priority queue effectively the design of bucketed integer priority queues over a finite range of rank values $[0, C]$ with number of buckets N, each covering C/N interval of the range. The number of buckets, and consequently the range covered by each bucket, depend on the required ranking granularity which is a characteristic of the scheduling policy. The number of buckets is typically in the range of a few thousands to hundreds of thousands. Elements falling within a range of a bucket are ordered in FIFO fashion. Theoretical complexity results for such bucketed integer priority queues are reported in [53, 29, 52].

Integer priority queues do not come for free. Efficient implementation of integer priority queues requires pre-allocation of buckets and meta data to access those buckets. In a packet scheduling setting the number of buckets is fixed, making the overhead per packet a constant whose value is logarithmic in the number of buckets, because searching is performed on the bucket list not the list of elements. Hence, bucketed integer priority queues achieve CPU efficiency at the expense of maintaining elements unsorted within a single bucket and pre-allocation of memory for all buckets. Note that the maintaining elements unsorted within a bucket is inconsequential because packets within a single bucket effectively have equivalent rank. Moreover, the memory required for buckets, in most cases, is minimal (e.g., tens to hundreds of kilobytes), which is consistent with earlier work on bucketed queues [47]. Another advantage of bucketed integer priority queues is that elements can be (re)moved with $O(1)$ overhead. This operation is used heavily in several scheduling algorithms (e.g., hClock [19] and pFabric [14]).

Recently, there has been some attempts to employ data structures specifically developed or re-purposed for efficiently implementing specific packet scheduling algorithms. For instance, Carousel [47], a system developed for rate limiting at scale, relies on Timing Wheel [54], a data structure that can support time-based operations in $O(1)$ and requires comparable memory to our proposed approach. However, Timing Wheel supports only non-work conserving time-based schedules in $O(1)$. Timing Wheel is efficient as buckets are indexed based on time and elements are accessed when their deadline arrives. However, Timing Wheel does not support operations needed for non-work conserving schedules (i.e., `ExtractMin` or `ExtractMax`). Another example is efficient approximation of popular scheduling policies (e.g., Start-Time Fair Queueing [31] as an approximation of Weighted Fair Queuing [24], or the more recent Quick Fair Queue (QFQ) [22]). This approach of developing a new system or a new data structure per scheduling policy does not provide a path to the efficient implementation of more complex policies. Furthermore, it does not allow for a truly programmable network. These limitations lead us to our first objective for Eiffel:

Objective 1: Develop data structures that can be employed for any scheduling algorithm providing $O(1)$ processing overhead per packet leveraging integer priority queues (§3.1).

Flexibility of Programmable Packet Schedulers: There has been recent interest in developing flexible, programmable, packet schedulers [50, 39]. This line of work is motivated by the support for programmability in all aspects of modern networks. Work on programmable schedulers focuses on providing the infrastructure for network operators to define their own scheduling policies. This approach improves on the current standard approach of providing a small fixed set of scheduling policies as currently provided in modern switches. A programmable scheduler provides building blocks for customizing packet ranking and transmission timing. Proposed programmable schedulers differ based on the flexibility of their building blocks. A flexible scheduler allows a network operator to specify policies according to the following specifications:

- *Unit of Scheduling:* Scheduling policies operate either on per packet basis (e.g., pacing) or on per flow basis (e.g., fair queuing). This requires a model that provides abstractions for both.
- *Work Conservation:* Scheduling policies can be work-conserving or non-work-conserving.
- *Ranking Trigger:* Efficient implementation of policies can require ranking packets on their enqueue, dequeue, or both.

Recent programmable packet schedulers export primitives that enable, the specification of a scheduling policy and its

parameters, often within limits. The PIFO scheduler programming model is the most prominent example [50]. It is implemented in hardware relying on Push-In-First-Out (PIFO) building blocks where packets are ranked only on enqueue. The scheduler is programmed by arranging the blocks to implement different scheduling policies. Due to its hardware implementation, the PIFO model employs compact constructs with considerable flexibility. However, PIFO remains very limited in its capacity (i.e., PIFO can handle a a maximum of 2048 flows at line rate), and expressiveness (i.e., PIFO can't express per flow scheduling). OpenQueue is an example of a flexible programmable packet scheduler in software [39]. However, the flexibility of OpenQueue comes at the expense of having three of its building blocks as priority queues, namely queues, buffers, and ports. This overhead, even in the presence of efficient priority queues, will form a memory and processing overhead. Furthermore, OpenQueue does not support non-work-conserving schedules.

The design of a flexible and efficient packet scheduler remains an open research challenge. It is important to note here that the efficiency of programmable schedulers is different from the efficiency of policies that they implement. An efficient programmable platform aims to reduce the overhead of its building blocks (i.e., Objective 1) which makes the overhead primarily a function of the complexity of the policy itself. Thus, the efficiency of a scheduling policy becomes a function of only the number of building blocks required to implement it. Furthermore, an efficient programmable platform should allow the operator to choose policies based on their requirements and available resources by allowing the platform to capture a wide variety of policies. To address this challenge, we choose to extend the PIFO model due to its existing efficient building blocks. In particular, we introduce flows as a unit of scheduling in the PIFO model. We also allow modifications to packet ranking and relative ordering both on enqueue and dequeue.

Objective 2: Provide a fully expressive scheduler programming abstraction by extending the PIFO model (§3.2).

Eiffel's place in Scheduling Research Landscape: This section reviewed scheduling support in software [1]. Table 1 summarizes the discussed related work. Eiffel fills the gap in earlier work by being the first efficient $O(1)$ and programmable software scheduler. It can support both per flow policies (e.g., hClock and pFabric) and per packet scheduling policies (e.g., Carousel). It can also support both work-conserving and non-work-conserving schedules.

[1]Scheduling is widely supported in hardware switches using a short list of scheduling policies, including shaping, strict priority, and Weighted Round Robin [7, 6, 9, 50]. An approach to efficient hardware packet scheduling relies on pipelined-heaps [18, 38, 55] to help position Eiffel. Pipelined-heaps are composed of piplined-stages for enqueuing and dequeuing elements in a priority queue. However, such approaches are not immediately applicable to software.

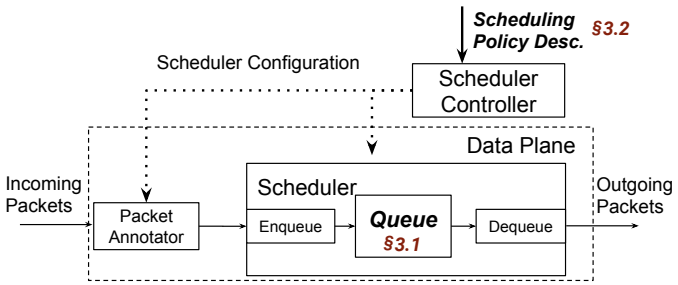

Figure 1: Eiffel programmable scheduler architecture highlighting Eiffel's extensions.

3 Eiffel Design

Figure 1 shows the architecture of Eiffel with four main components: 1) a packet annotator to set the input to the enqueue component (e.g., packet priority), 2) an enqueue component that calculates a rank for incoming packets, 3) a queue that holds packets sorted based on their rank, and 4) a dequeue component which is triggered to re-rank elements in the queue, for some scheduling algorithms. Eiffel leverages and extends the PIFO scheduler programming model to describe scheduling policies [50, 2]. The functions of the packet annotator, the enqueue module, and the dequeue module are derived in a straightforward manner from the scheduling policy. The only complexity in the Scheduler Controller, namely converting scheduling policy description to code, has been addressed in earlier work on the PIFO model [2]. The two complicated components in this architecture, therefore, correspond with the two objectives discussed in the previous section: the Queue *(Objective 1)* and The Scheduling Policy Description *(Objective 2)*. For the rest of this section, we explain our efficient queue data structures along with our extensions to the programming model used to configure the scheduler.

3.1 Priority Queueing in Eiffel

A priority queue maintains a list of elements, each tagged with a priority value. A priority queue supports one of two operations efficiently: `ExtractMin` or `ExtractMax` to get the element with minimum or maximum priority respectively. Our goal, as stated in Objective 1 in the previous section, is to enable these operations with $O(1)$ overhead. To this end we first develop a circular extension of efficient priority queues that rely on the FindFirstSet (FFS) operation, found in all modern CPUs [3, 10]. Our extensions allow FFS-based queues to operate over large moving ranges while maintaining CPU efficiency. We then improve on the FFS-based priority queue by introducing the approximate gradient queue, which can perform priority queuing in $O(1)$ under some conditions. The approximate priority queue can outperform the FFS-based queue by up to 9% for scenarios of a highly occupied bucketed priority queue (§5.2). Note

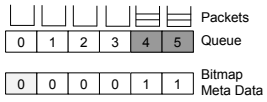

Figure 2: FFS-based queue where FFS of a bit-map of six bits can be processed in O(1).

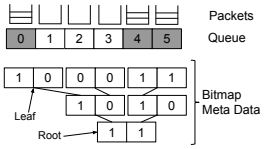

Figure 3: Hierarchical FFS-based queue where FFS of bit-map of two bits can be processed in O(1) using a 3-level hierarchy

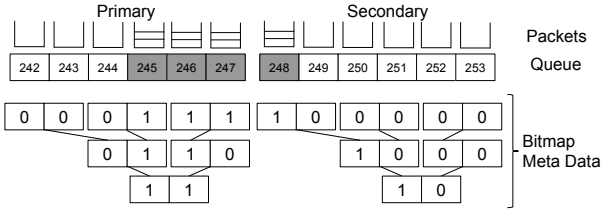

Figure 4: Circular Hierarchical FFS-based queue is composed of two Hierarchical FFS-based queues, one acting as the main queue and the other as a buffer.

that for all Integer Priority Queues discussed in this section, enqueue operation is trivial as buckets are identified by the priority value of their elements. This makes the enqueue operation a simple bucket lookup based on the priority value of the enqueued element.

3.1.1 Circular FFS-based Queue (cFFS)

FFS-based queues are bucketed priority queues with a bitmap representation of queue occupancy. Zero represents an empty bucket, and one represents a non-empty bucket. FFS produces the index of the leftmost set bit in a machine word in constant time. All modern CPUs support a version of Find First Set at a very low overhead (e.g., Bit-Scan-Forward (BSR) takes three cycles to complete [3]). Hence, a priority queue, with a number of buckets equal to or smaller than the width of the word supported by the FFS operation can obtain the smallest set bit, and hence the element with the smallest priority, in $O(1)$ (e.g., Figure 2). In the case that a queue has more buckets than the width of the word supported by a single FFS operation, a set of words can be processed sequentially to represent the queue, with every bit representing a bucket. This results in an $O(M)$ algorithm that is very efficient for very small M, where M is the number of words. For instance, realtime process scheduling in the linux kernel has a hundred priority levels. An FFS-based priority queue is used where FFS is applied sequentially on two words, in case of 64-bit words, or four words in case of 32-bit words [11]. This algorithm is not efficient for large values of M as it requires scanning all words, in the worst case, to find the index of the highest priority element. FFS instruction is also used in QFQ to sort groups of flows based on the eligibility for transmission where the number of groups is limited to a number smaller than 64 [22]. QFQ is an efficient implementation of fair queuing which uses FFS efficiently over a small number of elements. However, QFQ does not provide any clear direction towards implementing other policies efficiently.

To handle an even larger numbers of priority levels, hierarchical bitmaps may be used. One example is Priority Index Queue (PIQ) [55], a hardware implementation of FFS-based queues, which introduces a hierarchical structure where each node represents the occupancy of its children, and the children of leaf nodes are buckets. The minimum element can be found by recursively navigating the tree using FFS operation (e.g., Figure 3 for a word width of two). Hierarchical FFS-based queues have an overhead of $O(\log_w N)$ where w is the width of the word that FFS can process in $O(1)$ and N is the number of buckets. It is important to realize that, for a given scheduling policy, the value of N is a given fixed value that doesn't change once the scheduling policy is configured. Hence, a specific instance of a Hierarchical FFS-based queue has a constant overhead independent of the number of enqueued elements. In other words, once an implementation is created N does not change.

Hierarchical FFS-based queues only work for a fixed range of priority values. However, as discussed earlier, typical priority values for packets span a moving range. PIQ avoids this problem by assuming support for the universe of possible values of priorities. This is an inefficient approach because it requires generating and maintaining a large number of buckets, with relatively few of them in use at any given time.

Typical approaches to operating over a large moving range while maintaining a small memory footprint rely on *circular queues*. Such queues rely on the *mod* operation to map the moving range to a smaller range. However, the typical approach to circular queuing does not work in this case as it results in an incorrect bitmap. For example, if we add a packet with priority value six to the queue in Figure 2 selecting the bucket with a *mod* operation, the packet will be added in slot zero and consequently mark the bit map at slot zero. Hence, once the range of an FFS-based queue is set, all elements enqueued in that range have to be dequeued before the queue can be assigned a new range so as to avoid unnecessary resetting of elements. In that scenario, enqueued elements that are out of range are enqueued at the last bucket, and thus losing their proper ordering. Otherwise, the bitmap meta data will have to be reset in case any changes are made to the range of the queue.

A natural solution to this problem is to introduce an overflow queue where packets with priority values outside the current range are stored. Once all packets in the current range are dequeued, packets from that "secondary" queue are inserted using the new range. However, this introduces a significant overhead as we have to go through all pack-

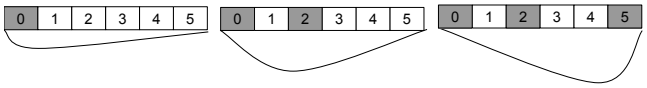

Figure 5: A sketch of a curvature function for three states of a **maximum** priority queue. As the maximum index of nonempty buckets increases, the critical point shifts closer to that index.

ets in the buffer every time the range advances. We solve this problem by making the secondary queue an FFS-based queue, covering the range that is immediately after the range of the queue (Figure 4). Elements outside the range of the secondary queue are enqueued at the last bucket in the secondary queue and their values are not sorted properly. However, we find that to not be a problem as ranges for the queues are typically easy to figure out given a specific scheduling policy.

A *Circular Hierarchical FFS-based queue*, referred to hereafter simply as a cFFS, maintains the minimum priority value supported by the primary queue (`h_index`), the number of buckets (`q_size`) per queue, two pointers to the two sets of buckets, and two pointers to the two sets of bitmaps. Hence, the queue "circulates" by switching the pointers of the two queues from the buffer range to the primary range and back based on the location of the minimum element along with their corresponding bitmaps.

Note that work on efficient priority queues has a very long history in computer science with examples including van Emde Boas tree [53] and Fusion trees [29]. However, such theoretical data structures are complicated to implement and require complex operations. cFFS is highly efficient both in terms of complexity and the required bit operations. Moreover, it is relatively easy to implement.

3.1.2 Approximate Priority Queuing

cFFS queues still require more than one step to find the minimum element. We explore a tradeoff between accuracy and efficiency by developing a gradient queue, a data structure that can find a *near* minimum element in one step.

Basic Idea The Gradient Queue (GQ) relies on an algebraic approach to calculating FFS. In other words, it attempts to find the index of the most significant bit using algebraic calculations. *This makes it amenable to approximation*. The intuition behind GQ is that the contribution of the most significant set bit to the value of a word is larger than the sum of the contributions of the rest of the set bits. We consider the weight of a non-empty bucket to be proportional to its index. Hence, Gradient Queue occupancy is represented by its curvature function. The curvature function of the queue is the sum of the weight functions of all nonempty buckets in the queue. More specifically, a specific curvature shape corresponds to a specific occupancy pattern. *A proper weight function* ensures the uniqueness of the curvature function per occupancy pattern. It also makes finding the non-empty bucket with the maximum index equivalent to finding the critical point of the queue's curvature (i.e., the point where the derivative of the curvature function of the queue is zero). A sample sketch of a curvature function is illustrated in Figure 5.

Exact Gradient Queue On a bucket becoming nonempty, we add its weight function to the queue's curvature function, and we subtract its function when it becomes empty. We define a desirable weight function as one that is: 1) easy to differentiate to find the critical point, and 2) easy to maintain when bucket state changes between empty and non-empty. We use weight function, $2^i(x-i)^2$ where i is the index of the bucket and x is the variable in the space of the curvature function.

This weight function results in queue curvature of the form of ax^2-bx+c, where the critical point is located at $x=b/2a$. Hence, we only care about a and b where $a=\sum_i 2^i$ and $b=\sum_i i2^i$ for all non-empty buckets i. The maintenance of the curvature function of the queue becomes as simple as incrementing and decrementing a and b when a bucket becomes non-empty or empty respectively. Theorem 1, in Appendix A, shows that determining the highest priority nonempty queue can be calculated using $ceil(b/a)$.

A Gradient Queue with a single curvature function is limited by the the range of values a and b can take, which is analogous to the limitation of FFS-based queues by the size of words for which FFS can be calculated in $O(1)$. A natural solution is to develop a hierarchical Gradient Queue. This makes Gradient Queue an equivalent of FFS-based queue with more expensive operations (i.e., division is more expensive than bit operations). However, due to its algebraic nature, Gradient Queue allows for approximation that is not feasible using bit operations.

Approximate Gradient Queue Like FFS-based queues, gradient queue has a complexity of $O(\log_w N)$ where w is the width of the representation of a and b and N is the number of buckets. Our goal is reduce the number of steps even further for each lookup. We are particularly interested in having lookups that can be made in one operation, which can be achieved through approximation. The advantage of the curvature representation of the Gradient Queue compared to FSS-based approaches is that it lends itself naturally to approximation.

A simple approximation is to make the value of a and b corresponding to a certain queue curvature smaller which will allow them to represent a larger number of priority values. In particular, we change the weight function to $2^{f(i)}(x-i)^2$ which results in $a=\sum_i 2^{f(i)}$ and $b=\sum_i i2^{f(i)}$ where $f(i)=i/\alpha$ and α is a positive integer. This approach leads to two natural results: 1) the biggest gain of the approximation is that a and b can now represent a much larger range of values for i which eliminates the need for hierarchical Gradient Queue and allows for finding the minimum ele-

ment with one step, and 2) the employed weight function is no longer proper. While BSR instruction is 8-32x faster than DIV [3], the performance gained from the reduced memory lookups required per BSR operation.

This approximation stems from using an "improper" weight function. This leads to breaking the two guarantees of a proper weight function, namely: 1) the curvature shape is no longer unique per queue occupancy pattern, and 2) the index of the maximum non-empty bucket no longer corresponds to the critical point of the curvature *in all cases*. In other words, the index of the maximum non-empty bucket, M, is no longer $ceil(b/a)$ due the fact that the weight of the maximum element no longer dominates the curvature function as the growth is sub-exponential. However, this ambiguity does not exist for all curvatures (i.e., queue occupancy patterns).

We characterize the conditions under which ambiguity occurs causing error in identifying the highest priority nonempty bucket. Hence, we identify scenarios where using the approximate queue is acceptable. The effect of $f(i) = i/\alpha$ can be described as introducing ambiguity to the value of $ceil(b/a)$. This is because exponential growth in a and b occurs not between consecutive indices but every α indices. In particular, we find solving the geometric and arithmetic-geometric sums of a and b that $\frac{b}{a} = \frac{M}{1-g(\alpha,M)} + u(\alpha)$ where $g(\alpha,M) = (2^{1/\alpha})^{-M-1}$ is a logarithmically decaying function of M and α. $u(\alpha) = 1/(1-2^{1/\alpha})$ is non-linear but slowly growing function of α. Hence, an approximate GQ can operate as a bucketed-queue where indices start from I_0 where $g(\alpha,M_0) \approx 0$ and end at I_{max} where $2^{f(I_{max})}$ can be precisely represented in the CPU word used to represent a and b. In this case, there is a constant shift in the value $ceil(b/a)$ that is calculated by $u(\alpha)$. For instance, consider an approximate queue with an α of 16. The function $g(\alpha,M)$ decays to near zero at $M = 124$ making the shift $u(\alpha) = 22$. Hence, $I_0 = 124$ and $I_{max} = 647$ which allows for the creation of an approximate queue that can handle 523 buckets. Note that this configuration results in an exact queue only when all buckets between I_0 and I_{max} are nonempty. However, error is introduced when some elements are missing. In Section 5.2, we show the effect of this error through extensive experiments; more examples are shown in Appendix B.

Typical scheduling policies (e.g., timestamp-based shaping, Least Slack Time First, and Earliest Deadline First) will generate priority values for packets that are uniformly distributed over priority levels. For such scenarios, the approximate gradient queue will have zero error and extract the minimum element in one step. This is clearly not true for *all* scheduling policies (e.g., strict priority will probably have more traffic for medium and low level priorities compared to high priority). For cases where the index suggested by the function is of an empty bucket, we perform linear search until we find a nonempty bucket. Moreover, for a cases of a moving range, a circular approximate queue can be implemented as with cFFS.

Approximate queues have been used before for different use cases. For instance, Soft-heap [21] is an approximate priority queue with a bounded error that is inversely proportional to the overhead of insertion. In particular, after n insertions in a soft-heap with an error bound $0 < \varepsilon \leq 1/2$, the overhead of insertion is $O(\log(1/\varepsilon))$. Hence, ExtractMin operation which can have a large error under Soft-heap. Another example is the RIPQ which is was developed for caching [51]. RIPQ relies on a bucket-sort-like approach. However, the RIPQ implementation is suited for static caching, where elements are not moved once inserted, which makes it not very suitable for the dynamic nature of packet scheduling.

3.2 Flexibility in Eiffel

Our second objective is to deploy flexible schedulers that have full expressive power to implement a wide range of scheduling policies. Our goal is to provide the network operator with a compiler that takes as input policy description and produces an initial implementation of the scheduler using the building blocks provided in the previous section. Our starting point is the work in PIFO which develops a model for programmable packet scheduling [50]. PIFO, however, suffers from several drawbacks, namely: 1) it doesn't support reordering packets already enqueued based on changes in their flow ranking, 2) it does not support ranking of elements on packet dequeue, and 3) it does not support shaping the output of the scheduling policy. In this section, we show our augmentation of the PIFO model to enable a completely flexible programming model in Eiffel. We address the first two issues by adding programming abstractions to the PIFO model, and we address the third problem by enabling arbitrary shaping with Eiffel by changing how shaping is handled within the PIFO model. We discuss the implementation of an initial version of the compiler in Section 4.

3.2.1 PIFO Model Extensions

Before we present our new abstractions, we review briefly the PIFO programming model [50]. The model relies on the Push-In-First-Out (PIFO) conceptual queue as its main building block. In programming the scheduler, the PIFO blocks are arranged to implement different scheduling algorithms.

The PIFO programming model has three abstractions: 1) scheduling transactions, 2) scheduling trees, and 3) shaping transactions. A scheduling transaction represents a single ranking function with a single priority queue. Scheduling trees are formed by connecting scheduling transactions, where each node's priority queue contains an ordering of its children. The tree structure allows incoming packets to change the relative ordering of packets belonging to different policies. Finally, a shaping transaction can be attached to any non-root node in the tree to enforce a rate limit on it.

There are several examples of the PIFO programming model in action presented in the original paper [50]. The primitives presented in the original PIFO model capture scheduling policies that have one of the following features: 1) distinct packet rank enumerations, over a small range of values (e.g., strict priority), 2) per-packet ranking over a large range of priority values (e.g., Earliest Deadline First [41]), and 3) hierarchical policy-based scheduling (e.g., Hierarchical Packet Fair Queuing [17]).

Eiffel augments the PIFO model by adding two additional scheduler primitives. The first primitive is *per-flow ranking and scheduling* where the rank of all packets of a flow depend on a ranking that is a function of the ranks of all packets enqueued for that specific flow. We assume that a sequence of packets that belong to a single flow should not be reordered by the scheduler. Existing PIFO primitives keep per-flow state but use them to rank each packet individually where an incoming packet for a certain flow does not change the ranking of packets already enqueued that belong to the same flow. The per-flow ranking extension keeps track of that information along with a queue per flow for all packets belonging to that flow. A single PIFO block orders flows, rather than packets, based on their rank. The second primitive is *on-dequeue scheduling* where incoming and outgoing packets belonging to a certain flow can change the rank of all packets belonging to that flow on enqueue and dequeue.

The two primitives can be integrated in the PIFO model. All flows belonging to a *per-flow* transaction are treated as a single flow by scheduling transactions higher in the hierarchical policy. Also note that every individual flow in the flow-rank policy can be composed of multiple flows that are scheduled according to per packet scheduling transactions. We realize that this specification requires tedious work to describe a complex policy that handles thousands of different flows or priorities. However, this specification provides a direct mapping to the underlying priority queues. We believe that defining higher level programming languages describing packet schedulers as well as formal description of the expressiveness of the language to be topics for future research.

3.2.2 Arbitrary Shaping

A flexible packet scheduler should support any scheme of bandwidth division between incoming flows. Earlier work on flexible schedulers either didn't support shaping at all (e.g., OpenQueue) or supported it with severe limitations (e.g., PIFO). We allow for arbitrary shaping by decoupling work conserving scheduling from shaping. A natural approach to this decoupling is to allow any flow or group of flows to have a shaper associated with them. This can be achieved by assigning a separate queue to the shaped aggregate whose output is then enqueued into its proper location in the scheduling hierarchy. However, this approach is extremely inefficient as it requires a queue per rate limit, which can lead to increased CPU and memory overhead. We improve the efficiency of this approach by leveraging recent results that show that any rate limit can be translated to a timestamp per packet, which yields even better adherence to the set rate than token buckets [47]. Hence, we use only one shaper for the whole hierarchy which is implemented using a single priority queue.

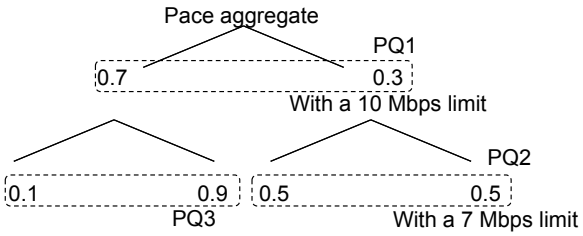

Figure 6: Example of a policy that imposes two limits on packets the belong to the rightmost leaf.

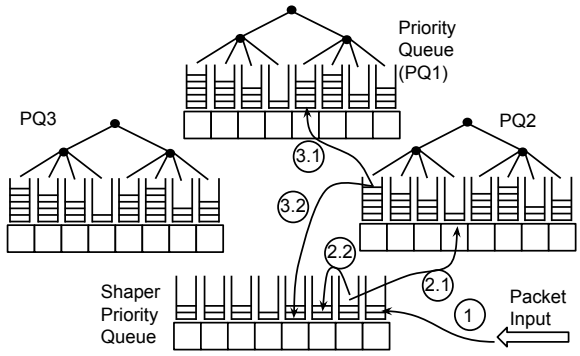

Figure 7: A diagram of the implementation of the example in Figure 6.

As an example, consider the hierarchical policy in Figure 6. Each node represents a policy-defined flow with the root representing the aggregate traffic. Each node has a share of its parent's bandwidth, defined by the fraction in the figure. Each node can also have a policy-defined rate limit. In this example, we have a rate limit at a non-leaf node and a leaf node. Furthermore, we require the aggregate traffic to be paced. We map the hierarchical policy in Figure 6 to its priority-queue-based realization in Figure 7. Per the PIFO model, each non-leaf node is represented by a priority queue. Per our proposal, a single shaper is added to rate limit all packets according to all policy-defined rate limits.

To illustrate how this single shaper works, consider packets belonging to the rightmost leaf policy. We explore the journey of packets belonging to that leaf policy through the different queues, shown in Figure 7. These packets will be enqueued to the shaper with timestamps set based on a 7 Mbps rate to enforce the rate on their node (step 1). Once dequeued from the shaper, each packet will be enqueued to PQ2 (step 2.1) and the shaper according to the 10 Mbps rate limit (step 2.2). After the transmission time of a packet belonging to PQ2 is reached, which is defined by the shaper, the packet is inserted in both the root's (PQ1) priority queue

(3.1) and the shaper according to the pacing rate (3.2). When the transmission time, calculated based on the pacing rate, is reached the packet is transmitted. To achieve this functionality, each packet holds a pointer to the priority queue they should be enqueued to. This pointer avoids searching for the queue a packet should be enqueued to. Note that having the separate shaper allows for specifying rate limits on any node in the hierarchical policy (e.g., the root and leaves) which was not possible in the PIFO model, where shaping transactions are tightly coupled with scheduling transactions.

4 Eiffel Implementation

Packet scheduling is implemented in two places in the network: 1) hardware or software switches, and 2) end-host kernel. We focus on the software placements (kernel and userspace switches) and show that Eiffel can outperform the state of the art in both settings. We find that userspace and kernel implementations of packet scheduling face significantly different challenges as the kernel operates in an event-based setting while userspace operates in a busy polling setting. We explain here the differences between both implementations and our approach to each. We start with our approach to policy creation.

Policy Creation: We extend the existing PIFO open source model to configure the scheduling algorithm [50, 2]. The existing implementation represents the policy as a graph using the DOT description language and translates the graph into C++ code. We rely on the cFFS for our implementation, unless otherwise stated. This provides an initial implementation which we tune according to whether the code is going to be used in kernel or userspace. We believe automating this process can be further refined, but the goal of this work is to evaluate the performance of Eiffel algorithms and data structures.

Kernel Implementation We implement Eiffel as a qdisc [36] kernel module that implements enqueue and dequeue functions and keeps track of the number of enqueued packets. The module can also set a timer to trigger dequeue. Access to qdiscs is serialized through a global qdisc lock. In our design, we focus on two sources of overhead in a qdisc: 1) the overhead of the queuing data structure, and 2) the overhead of properly setting the timer. Eiffel reduces the first overhead by utilizing one of the proposed data structures to reduce the cost of both enqueue and dequeue operations. The second overhead can be mitigated by improving the efficiency of finding the smallest deadline of an enqueued packet. This operation of `SoonestDeadline()` is required to efficiently set the timer to wake up at the deadline of the next packet. Either of our supported data structures can support this operation efficiently as well.

Userspace Implementation We implement Eiffel in the Berkeley Extensible Software Switch (BESS, formerly SoftNIC [33]). BESS represents network processing elements as a pipeline of modules. BESS is busy polling-based where a set of connected modules form a unit of execution called a task. A scheduler tracks all tasks and runs them according to assigned policies. Tasks are scheduled based on the amount of resources (CPU cycles or bits) they consume. Our implementation of Eiffel in BESS is done in self-contained modules.

We find that two main parameters determine the efficiency of Eiffel in BESS: 1) batch size and 2) queue size. Batching is already well supported in BESS as each module receives packets in batches and passes packets to its subsequent module in a batch. However, we find that batching per flow has an intricate impact on the performance of Eiffel. For instance, with small packet sizes, if no batching is performed per flow, then every incoming batch of packets will activate a large number of queues without any of the packets being actually queued (due to small packet size) which increases the overhead per packet (i.e., queue lookup of multiple queues rather than one). This is not the case for large packet sizes where the lookup cost is amortized over the larger size of the packet improving performance compared to batching of large packets. Batching large packets results in large queues for flows (i.e., large number of flows with large number of enqueued packets). We find that batching should be applied based on expected traffic pattern. For that purpose, we setup `Buffer` modules per traffic class before Eiffel's module in the pipeline when needed. We also perform output batching per flow in units of 10KB worth of payload which was suggested as a good threshold that does not affect fairness at a macroscale between flows [19]. We also find that limiting the number of packets enqueued in Eiffel can significantly affect the performance of Eiffel in BESS. We limit the number of packets per flow to 32 packets which we find, empirically, to maintain performance.

5 Evaluation

5.1 Eiffel Use Cases

Methodology: We evaluate our kernel and userspace implementation through a set of use cases each with its corresponding baseline. We implement two common use cases, one in kernel and another in userspace[2]. In each use case, we evaluated Eiffel's scheduling behavior as well as its CPU performance as compared to the baseline. The comparison of scheduling behavior was done by comparing aggregate rates achieved as well as order of released packets. However, we only report CPU efficiency results as we find that Eiffel matches the scheduling behavior of the baselines.

A key aspect of our evaluation is determining the metrics of comparisons in kernel and userspace settings. The main difference is that a kernel module can support line rate by using more CPU. This requires us to fix the packet rate we are evaluating at and look at the CPU utilization of different

[2] A third use case the implements hClock in userspace can be found in the extended version of this paper [48]

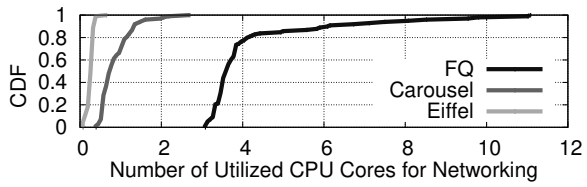

Figure 8: A comparison between the CPU overhead of the networking stack using FQ/pacing, Carousel, and Eiffel.

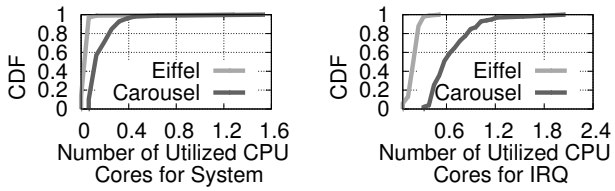

Figure 9: A Comparison between detailed CPU utilization of Carousel and Eiffel in terms of system processes (left) and soft interrupt servicing (right).

scheduler implementations. On the other hand, a userspace implementation relies on busy polling on one or more CPU cores to support different packet rates. Hence, in the case of userspace, we fix the number of cores used, to one core unless otherwise is stated, and compare the different scheduler implementations based on the maximum achievable rate.

5.1.1 Use Case 1: Shaping in Kernel

Traffic shaping (i.e., rate limiting and pacing) is an essential operation for efficient utilization [16] and correct operation of modern protocols (e.g., both TIMELY [43] and BBR [20] require per flow pacing). Recently, it has been shown that the canonical kernel shapers (i.e., FQ/pacing [26] and HTB qdiscs [25]) are inefficient due to reliance on inefficient data structures; there are outperformed by the userspace-based implementation in Carousel [47]. To offer a fair comparison we implement all systems in the kernel. We implement a rate limiting qdisc whose functionality matches the rate limiting features of the existing FQ/pacing qdisc [26].

We implemented Eiffel as a qdisc. The queue is configured with 20k buckets with a maximum horizon of 2 seconds and only the shaper is used. We implemented the qdisc in kernel v4.10. We modified only `sock.h` to keep the state of each socket allowing us to avoid having to keep track of each flow in the qdisc. We conduct experiments for egress traffic shaping between two servers within the same cluster in Amazon EC2. We use two `m4.16xlarge` instances equipped with 64 cores and capable of sustaining 25 Gbps. We use `neper` [5] to generate traffic with a large number of TCP flows. In particular, we generate traffic from 20k flows and use `SO_MAX_PACING_RATE` to rate limit individual flows to achieve a maximum aggregate rate of 24 Gbps. This configuration constitutes a worst case in terms of load for all evaluated qdiscs as it requires the maximum amount of calculations. We measure overhead in terms of the number of cores used for network processing which we calculate based on the observed fraction of CPU utilization. Without `neper` operating, CPU utilization is zero, hence, we attribute any CPU utilization during our experiments to the networking stack, except for the CPU portion attributed to userspace processes. We track CPU utilization using `dstat`. We run our experiments for 100 seconds and record the CPU utilization every second. This continuous behavior emulates the behavior handled by content servers which were used to evaluate Carousel [47].

```
#On enqueue of packet p of flow f:
f.rank = min(p.rank, f.rank)
#On dequeue of packet p of flow f:
f.rank = min(p.rank, f.front().rank)
```

Figure 10: Implementation of pFabric in Eiffel.

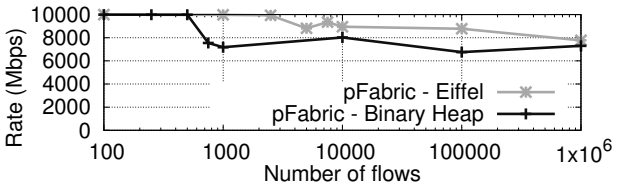

Figure 11: Performance of pFabric implementation using cFFS and a binary heap showing Eiffel sustaining line rate at 5x number of flows.

Figure 8 shows the overhead of all three systems. It is clear that Eiffel is superior, outperforming FQ by a median 14x and Carousel by 3x. We find the overhead of FQ to be consistent with earlier results [47]. This is due to its complicated data structure which keeps track internally of active and inactive flows and requires continuous garbage collection to remove old inactive flows. Furthermore, it relies on RB-trees which increases the overhead of reordering flows on every enqueue and dequeue. To better understand the comparison with Carousel, we look at the breakdown of the main components of CPU overhead, namely overhead spent on *system processes* and *servicing software interrupts*. Figure 9 details the comparison. We find that the main difference is in the overhead introduced by Carousel in firing timers at constant intervals while Eiffel can trigger timers exactly when needed (Figure 9 right). The overhead of the data structures in both cases introduces minimal overhead in system processes (Figure 9 left).

5.1.2 Use Case 2: Least/Largest X First in Userspace

One of the most widely used patterns for packet scheduling is ordering packets such that the flow or packet with the least or most of some feature exits the queue first. Many examples of such policies have been promoted including Least Slack Time First (LSTF) [42], Largest Queue First (LQF), and Shortest/Least Remaining Time First (SRTF). We refer to this class of algorithms as L(X)F. This class of algorithms

is interesting as some of them were shown to provide theoretically proven desirable behavior. For instance, LSTF was shown be a universal packet scheduler that can emulate the behavior of any scheduling algorithm [42]. Furthermore, SRTF was shown to schedule flows close to optimally within the pFabric architecture [14]. We show that Eiffel can improve the performance of this class of scheduling algorithms.

We implement pFabric as an instance of such class of algorithms where flows are ranked based on their remaining number of packets. Every incoming and outgoing packet changes the rank of all other packets belonging to the same flow, requiring on dequeue ranking. Figure 10 shows the representation of pFabric using the PIFO model with per-flow ranking and on dequeue ranking provided by Eiffel. We also implemented pFabric using $O(\log n)$ priority queue based on a Binary Heap to provide a baseline. Both designs were implemented as queue modules in BESS. We used packets of size 1500B. Operations are all done on a single core with a simple flow generator. All results are the average of ten experiments each lasting for 20 seconds. Figure 11 shows the impact of increasing the number of flows on the performance of both designs. It is clear that Eiffel has better performance. The overhead of pFabric stems from the need to continuously move flows between buckets which has $O(1)$ using bucketed queues while it has an overhead of $O(n)$ as it requires reheapifying the heap every time. The figure also shows that as the number of flows increases the value of Eiffel starts to decrease as Eiffel reaches its capacity.

5.2 Eiffel Microbenchmark

Our goal in this section is evaluate the impact of different parameters on the performance of different data structures. We also evaluate the effect of approximation in switches on network-wide objectives. Finally, we provide guidance on how one should choose among the different queuing data structures within Eiffel, given specific scheduler user-case characteristics. To inform this decision we run a number of microbenchmark experiments. We start by evaluating the performance of the proposed data structures compared to a basic bucketed priority queue implementation. Then, we explore the impact of approximation using the gradient queue both on a single queue and at a large network scale through ns2-simulation. Finally, we present our guide for choosing a priority queue implementation.

Experiment setup: We perform benchmarks using Google's benchmark tool [12]. We develop a baseline for bucketed priority queues by keeping track of non-empty buckets in a binary heap, we refer to this as BH. We ignore comparison-based priority queues (e.g., Binary Heaps and RB-trees) as we find that bucketed priority queues performs 6x better in most cases. We compare cFFS, approximate gradient queue (Approx), and BH. In all our experiments, the queue is initially filled with elements according to queue occupancy rate or average number of packet per bucket parameters. Then, packets are dequeued from the queue. Reported results (i.e., y-axis of figures 12 and 13) are in terms of million packets per seconds.

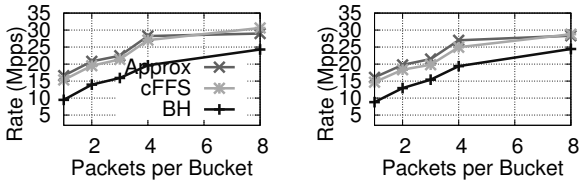

Figure 12: Effect of number of packets per bucket on queue performance for 5k (left) and 10k (right) buckets.

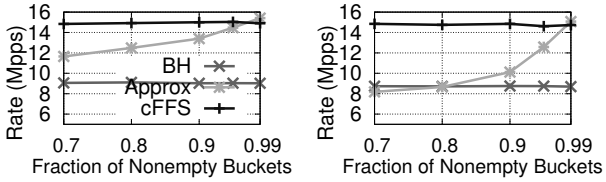

Figure 13: Effect of queue occupancy on performance of Approximate Queue for 5k (left) and 10k (right) buckets.

Effect of number of packet per bucket: The number of buckets configured in a queue is the main determining factor for the overhead of a bucketed queue. Note that this parameter controls queue granularity which is the priority interval covered by a bucket. High granularity (i.e., large number of buckets) implies a smaller number of packets per bucket for the same workload. Hence, the number of packets per bucket is a good proxy to the configured number of buckets. For instance, if we choose a large number of buckets with high granularity, the chance of empty buckets increases. On the other hand, if we choose a small number of buckets with coarser granularity, we get higher number of elements per bucket. This proxy is important because in the case of the approximate queue, the main factor affecting its performance is the number of empty buckets.

Figure 12 shows the effect of increasing the average number of packets per bucket for all three queues for 5k and 10k buckets. For a small number of packets per bucket, which also reflects choosing a fine grain granularity, the approximate queue introduces up to 9% improvement in performance in the case of 10k buckets. In such cases, the approximate queue function has zero error which makes it significantly better. As the number of the packets per bucket increases, the overhead of finding the smallest indexed bucket is amortized over the total number of elements in the bucket which makes FFS-based and approximate queues similar in performance.

We also explore the effect of having empty buckets on the performance of the approximate queue. Empty buckets cause errors in the curvature function of the approximate queue which in turn trigger linear search for non-empty buckets. Figure 12 shows throughput of the queue for different ratios

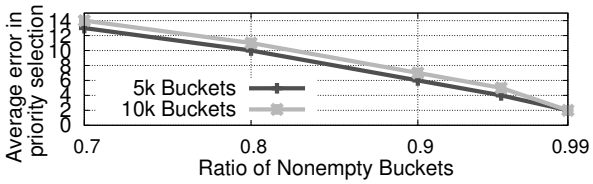

Figure 14: Effect of having empty buckets on the error of fetching the minimum element for the approximate queue.

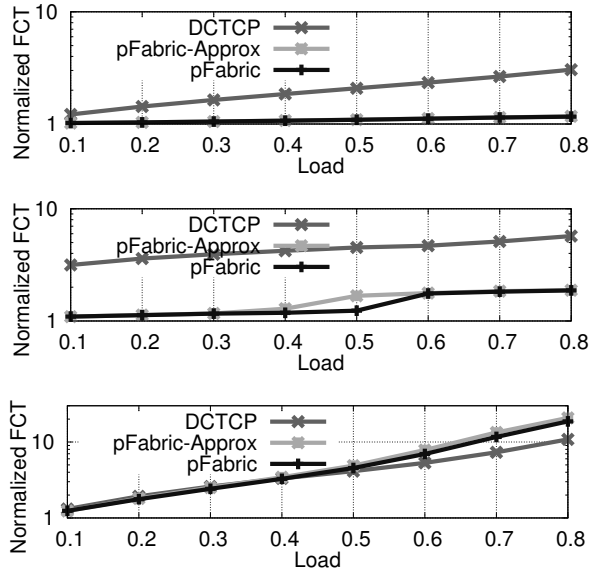

Figure 15: Effect of using an Approximate Queue on the performance of pFabric in terms of normalized flow completion times under different load characteristics: Average FCT for (0, 100kB] flow sizes, 99th percentile FCT for (0, 100kB] for sizes, and Average FCT for (10MB, inf) flow sizes.

of non-empty buckets. As expected, as the ratio increases the overhead decreases which improves the throughput of the approximate queue. Figure 14 shows the error in the approximate queue's fetching of elements. As the number of empty buckets increases the error in the approximate queue is larger and the overhead of linear search grows. We suggest that cases where the queue is more than 30% empty should trigger changes in the queue's granularity based on the queue's CPU performance and to avoid allocating memory to buckets that are not used.

The granularity of the queue determines the representation capacity of the queue. It is clear for our results that picking low granularity (i.e., high number of packets per bucket) yields better performance in terms of packets per second. On the other hand, from a networking perspective, high granularity yields exact ordering of packets. For instance, a queue with a granularity of 100 microseconds cannot insert gaps between packets that are smaller than 100 microseconds. Hence, we recommend configuring the queue's granularity such that each bucket has at least one packet. This

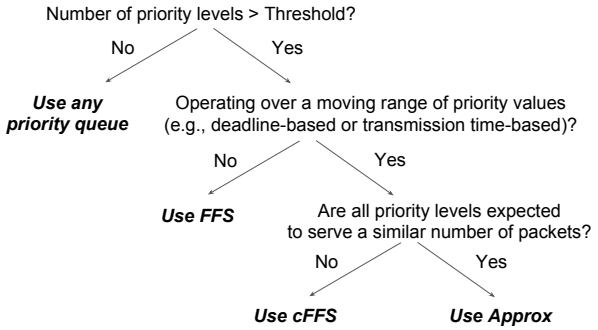

Figure 16: Decision tree for selecting a priority queue based on the characteristics of the scheduling algorithm.

can be determined by observing the long term behavior of the queue. We also note that this problem can be solved by having non-uniform bucket granularity which is dynamically set to achieve the result of at least one packet per bucket. We leave this problem for future work.

Impact of Approximation on Network-wide Objectives: A natural question is: how does approximate prioritization, at *every* switch in a network, affect network-wide objectives? To answer that question, we perform simulations of pFabric, which requires prioritization at every switch. Our simulation are based on `ns2` simulations provided by the authors of pFabric [14] and the plotting tools provided by the authors of QJump [32]. We change only the priority queuing implementation from a linear search-based priority queue to our Approximate priority queue and increase queue size to handle 1000k elements. We use DCTCP [13] as a baseline to put the result in context. Figure 15 shows a snapshot of results of the simulations of a 144 node leaf-spine topology. Due to space limitations, We show results for only web-search workload simulations which are based on clusters in Microsoft datacenters [13]. The load is varied between 10% to 80% of the load observed. We note that the setting of the simulations is not relevant for the scope of this paper, however, what is relevant is comparing the performance of pFabric using its original implementation to pFabric using our approximate queue. We find that approximation has minimal effect on overall network behavior which makes performance on a mircorscale the only concern in selecting a queue for a specific scheduler.

A Guide for Choosing a Priority Queue for Packet Scheduling Figure 16 summarizes our takeaways from working with the proposed queues. For a small number of priority levels, we find that the choice of priority queue has little impact and for most scenarios a bucket-based queue might be overkill due to its memory overhead. However, when the number of priority levels or buckets is larger than a threshold the choice of queues makes a significant difference. We found in our experiments that this threshold is 1k and that the difference in performance is not significant around the threshold. We find that if the priority levels are

over a fixed range (e.g., job remaining time [14]) then an FFS-based priority queue is sufficient. When the priority levels are over a moving range, where the number of levels are not all equally likely (e.g., rate limiting with a wide range of limits [47]), it is better to use cFFS priority queue. However, for priority levels over a moving range with highly occupied priority levels (e.g., Least Slack Time-based [42] or hierarchical-based schedules [19]) approximate queue can be beneficial.

Another important aspect is choosing the number of buckets to assign to a queue. This parameter should be chosen based on both the desired granularity and efficiency which form a clear trade-off. Proposed queues have minimal CPU overhead (e.g., a queue with a billion buckets will require six bit operations to find the minimum non-empty bucket using a cFFS). Hence, the main source of efficiency overhead is the memory overhead which has two components: 1) memory footprint, and 2) cache freshness. However, we find that most scheduling policies require thousands to tens of thousands of elements which require small memory allocation for our proposed queues.

6 Conclusion

Efficient packet scheduling is a crucial mechanism for the correct operation of networks. Flexible packet scheduling is a necessary component of the current ecosystem of programmable networks. In this paper, we showed how Eiffel can introduce both efficiency and flexibility for packet scheduling in software relying on integer priority queuing concepts and novel packet scheduling programming abstractions. We showed that Eiffel can achieve orders of magnitude improvements in performance compared to the state of the art while enabling packet scheduling at scale in terms of both number of flows or rules and line rate. We believe that our work should enable network operators to have more freedom in implementing complex policies that correspond to current networks needs where isolation and strict sharing policies are needed.

We believe that the biggest impact Eiffel will have is making the case for a reconsideration of the basic building blocks of the packet schedulers in hardware. Current proposals for packet scheduling in hardware (e.g., PIFO model [50] and SmartNICs [28]), rely on parallel comparisons of elements in a single queue. This approach limits the size of the queue. Earlier proposals that rely on pipelined-heaps [18, 38, 55] required a priority queue that can capture the whole universe of possible packet rank values, which requires significant hardware overhead. We see Eiffel as a step on the road of improving hardware packet schedulers by reducing the number of parallel comparisons through an FFS-based queue meta data or through an approximate queue metadata. For instance, Eiffel can be employed in a hierarchical structure with parallel comparisons to increase the capacity of individual queues in a PIFO-like setting. Future programmable schedulers can implement a hardware version of cFFS or the approximate queue and provide an interface that allows for connecting them according to programmable policies. While the implementation is definitely not straight forward, we believe this to be the natural next step in the development of scalable packet schedulers.

Acknowledgments

The authors would like to thank the NSDI Shepherd, K. K. Ramakrishnan, and the anonymous reviewers for providing excellent feedback. This work is funded in part by NSF grants NETS 1816331.

References

[1] IEEE Standard for Local and Metropolitan Area Networks—Virtual Bridged Local Area Networks. *IEEE Std 802.1Q-2005 (Incorporates IEEE Std 802.1Q1998, IEEE Std 802.1u-2001, IEEE Std 802.1v-2001, and IEEE Std 802.1s-2002)* (May 2006), 1–300.

[2] C++ reference implementation for Push-In First-Out Queue, 2016. https://github.com/programmable-scheduling/pifo-machine.

[3] Intel 64 and ia-32 architectures optimization reference manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2016.

[4] Intel 82599 10gbe controller. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf, 2016.

[5] neper: a Linux networking performance tool, 2016. https://github.com/google/neper.

[6] Arista 7010T Gigabit Ethernet Data Center Switches. https://www.arista.com/assets/data/pdf/Datasheets/7010T-48_Datasheet.pdf, 2017.

[7] Arista 7500 series data center switch. https://www.arista.com/assets/data/pdf/Datasheets/7500_Datasheet.pdf, 2017.

[8] Bess: Berkeley extensible software switch. https://github.com/NetSys/bess/wiki, 2017.

[9] Cisco: Understanding Quality of Service on the Catalyst 6500 Switch. https://www.cisco.com/c/en/us/products/collateral/switches/catalyst-6500-series-switches/white_paper_c11_538840.html, 2017.

[10] MD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions. https://support.amd.com/TechDocs/24594.pdf, 2017.

[11] Real-Time Scheduling Class (mapped to the SCHED_FIFO and SCHED_RR policies). https://elixir.bootlin.com/linux/latest/source/kernel/sched/rt.c#L1494, 2017.

[12] Benchmark Tools, 2018. https://github.com/google/benchmark.

[13] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM Conference* (New York, NY, USA, 2010), ACM, pp. 63–74.

[14] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM Conference* (New York, NY, USA, 2013), ACM, pp. 435–446.

[15] BAI, W., CHEN, L., CHEN, K., HAN, D., TIAN, C., AND WANG, H. Information-agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 455–468.

[16] BEHESHTI, N., GANJALI, Y., GHOBADI, M., MCKEOWN, N., AND SALMON, G. Experimental study of router buffer sizing. In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement* (New York, NY, USA, 2008), IMC '08, ACM, pp. 197–210.

[17] BENNETT, J. C. R., AND ZHANG, H. Hierarchical packet fair queueing algorithms. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 1996), SIGCOMM '96, ACM, pp. 143–156.

[18] BHAGWAN, R., AND LIN, B. Fast and scalable priority queue architecture for high-speed network switches. In *INFOCOM '00* (2000), pp. 538–547.

[19] BILLAUD, J.-P., AND GULATI, A. hclock: Hierarchical qos for packet scheduling in a hypervisor. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13, ACM, pp. 309–322.

[20] CARDWELL, N., CHENG, Y., GUNN, C. S., YEGANEH, S. H., AND JACOBSON, V. BBR: Congestion-Based Congestion Control. *ACM Queue 14, September-October* (2016), 20 – 53.

[21] CHAZELLE, B. The soft heap: an approximate priority queue with optimal error rate. *Journal of the ACM (JACM) 47*, 6 (2000), 1012–1027.

[22] CHECCONI, F., RIZZO, L., AND VALENTE, P. QFQ: Efficient packet scheduling with tight guarantees. *IEEE/ACM Transactions on Networking (TON) 21*, 3 (2013), 802–816.

[23] DALTON, M., SCHULTZ, D., ADRIAENS, J., AREFIN, A., GUPTA, A., FAHS, B., RUBINSTEIN, D., ZERMENO, E. C., RUBOW, E., DOCAUER, J. A., ALPERT, J., AI, J., OLSON, J., DECABOOTER, K., DE KRUIJF, M., HUA, N., LEWIS, N., KASINADHUNI, N., CREPALDI, R., KRISHNAN, S., VENKATA, S., RICHTER, Y., NAIK, U., AND VAHDAT, A. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, 2018), USENIX Association, pp. 373–387.

[24] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queueing algorithm. In *Symposium Proceedings on Communications Architectures &Amp; Protocols* (New York, NY, USA, 1989), SIGCOMM '89, ACM, pp. 1–12.

[25] DEVERA, M. Linux Hierarchical Token Bucket, 2003. http://luxik.cdi.cz/~devik/qos/htb/.

[26] DUMAZET, E., AND CORBET, J. Tso sizing and the fq scheduler. https://lwn.net/Articles/564978/, 2013.

[27] FEAMSTER, N., AND REXFORD, J. Why (and how) networks should run themselves. *arXiv preprint arXiv:1710.11583* (2017).

[28] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A., CHUNG, E., CHANDRAPPA, H. K., CHATURMOHTA, S., HUMPHREY, M., LAVIER, J., LAM, N., LIU, F., OVTCHAROV, K., PADHYE, J., POPURI, G., RAINDEL, S., SAPRE, T., SHAW, M., SILVA, G., SIVAKUMAR, M., SRIVASTAVA, N., VERMA, A., ZUHAIR, Q., BANSAL, D., BURGER, D., VAID, K., MALTZ, D. A., AND GREENBERG, A. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, 2018), USENIX Association, pp. 51–66.

[29] FREDMAN, M. L., AND WILLARD, D. E. Blasting through the information theoretic barrier with fusion trees. In *STOC' 90* (1990), pp. 1–7.

[30] GHODSI, A., SEKAR, V., ZAHARIA, M., AND STOICA, I. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 1–12.

[31] GOYAL, P., VIN, H. M., AND CHEN, H. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 1996), SIGCOMM '96, ACM, pp. 157–168.

[32] GROSVENOR, M. P., SCHWARZKOPF, M., GOG, I., WATSON, R. N. M., MOORE, A. W., HAND, S., AND CROWCROFT, J. Queues don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 1–14.

[33] HAN, S., JANG, K., PANDA, A., PALKAR, S., HAN, D., AND RATNASAMY, S. Softnic: A software nic to augment hardware. Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[34] HONG, C.-Y., CAESAR, M., AND GODFREY, P. B. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review 42*, 4 (Aug. 2012), 127–138.

[35] HONG, C.-Y., KANDULA, S., MAHAJAN, R., ZHANG, M., GILL, V., NANDURI, M., AND WATTENHOFER, R. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM Conference* (New York, NY, USA, 2013), ACM, pp. 15–26.

[36] HUBERT, B., GRAF, T., MAXWELL, G., VAN MOOK, R., VAN OOSTERHOUT, M., SCHROEDER, P., SPAANS, J., AND LARROY, P. Linux advanced routing & traffic control. In *Ottawa Linux Symposium* (2002), p. 213.

[37] INC., V. Performance Evaluation of Network I/O Control in VMware vSphere 6. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/network-ioc-vsphere6-performance-evaluation-white-paper.pdf, 2015.

[38] IOANNOU, A., AND KATEVENIS, M. G. H. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Transactions on Networking 15*, 2 (April 2007), 450–461.

[39] KOGAN, K., MENIKKUMBURA, D., PETRI, G., NOH, Y., NIKOLENKO, S., SIROTKIN, A. V., AND EUGSTER, P. A programmable buffer management platform. In *ICNP '17* (2017).

[40] KUMAR, A., JAIN, S., NAIK, U., RAGHURAMAN, A., KASINADHUNI, N., ZERMENO, E. C., GUNN, C. S., AI, J., CARLIN, B., AMARANDEI-STAVILA, M., ROBIN, M., SIGANPORIA, A., STUART, S., AND VAHDAT, A. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *Proceedings of the ACM SIGCOMM Conference* (New York, NY, USA, 2015), ACM, pp. 1–14.

[41] LIU, C. L., AND LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM) 20*, 1 (1973), 46–61.

[42] MITTAL, R., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Universal Packet Scheduling. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)* (Mar. 2016), pp. 501–521.

[43] MITTAL, R., LAM, V. T., DUKKIPATI, N., BLEM, E., WASSEL, H., GHOBADI, M., VAHDAT, A., WANG, Y., WETHERALL, D., AND ZATS, D. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the ACM SIGCOMM Conference* (2015), pp. 537–550.

[44] MUNIR, A., BAIG, G., IRTEZA, S. M., QAZI, I. A., LIU, A. X., AND DOGAR, F. R. Friends, not foes: Synthesizing existing transport strategies for data center networks. In *Proceedings of the ACM SIGCOMM Conference* (New York, NY, USA, 2014), ACM, pp. 491–502.

[45] PFAFF, B., PETTIT, J., KOPONEN, T., JACKSON, E., ZHOU, A., RAJAHALME, J., GROSS, J., WANG, A., STRINGER, J., SHELAR, P., AMIDON, K., AND CASADO, M. The design and implementation of open vswitch. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)* (2015).

[46] RADHAKRISHNAN, S., GENG, Y., JEYAKUMAR, V., KABBANI, A., PORTER, G., AND VAHDAT, A. SENIC: scalable NIC for end-host rate limiting. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)* (2014).

[47] SAEED, A., DUKKIPATI, N., VALANCIUS, V., LAM, T., CONTAVALLI, C., AND VAHDAT, A. Carousel: Scalable Traffic Shaping at End-Hosts. In *Proceedings of the ACM SIGCOMM Conference* (2017).

[48] SAEED, A., ZHAO, Y., DUKKIPATI, N., AMMAR, M., ZEGURA, E., HARRAS, K., AND VAHDAT, A. Eiffel: Efficient and Flexible Software Packet Scheduling. *arXiv preprint arXiv:1810.03060* (2018).

[49] SIVARAMAN, A., SUBRAMANIAN, S., AGRAWAL, A., CHOLE, S., CHUANG, S.-T., EDSALL, T., ALIZADEH, M., KATTI, S., MCKEOWN, N., AND BALAKRISHNAN, H. Towards programmable packet scheduling. In *HotNets-XIV* (2015).

[50] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable Packet Scheduling at Line Rate. In *Proceedings of the ACM SIGCOMM Conference* (2016), pp. 44–57.

[51] TANG, L., HUANG, Q., LLOYD, W., KUMAR, S., AND LI, K. Ripq: Advanced photo caching on flash for facebook. In *FAST* (2015), pp. 373–386.

[52] THORUP, M. Equivalence between priority queues and sorting. *J. ACM 54*, 6 (2007).

[53] VAN EMDE BOAS, P. Preserving order in a forest in less than logarithmic time. In *FOCS' 75* (1975), pp. 75–84.

[54] VARGHESE, G., AND LAUCK, T. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1987), SOSP '87, ACM, pp. 25–38.

[55] WANG, H., AND LIN, B. Per-flow queue management with succinct priority indexing structures for high speed packet scheduling. *IEEE Transactions on Parallel and Distributed Systems 24*, 7 (2013), 1380–1389.

[56] WANG, W., FENG, C., LI, B., AND LIANG, B. On the fairness-efficiency tradeoff for packet processing with multiple resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2014), CoNEXT '14, ACM, pp. 235–248.

A Gradient Queue Correctness

Theorem 1. *The index of the maximum non-empty bucket, N, is* $ceil(b/a)$.

Proof. We encode the occupancy of buckets by a bit string of length N where zeros represent empty buckets and ones represent nonempty buckets. The value of the bit string is the value of the critical point $x = \frac{b}{a}$ for queue represented by the bit of strings. We prove the theorem by showing an ordering between all bit strings, where the maximum value is N and the minimum value is larger than $N-1$. The minimum value is when all buckets are nonempty (i.e., all ones). In that case, $a = \sum_{i=1}^{N} 2^i$ and $b = \sum_{i=1}^{N} i2^i$. Note that b is an Arithmetic-Geometric Progression that can be simplified to $N2^{N+1} - (2^{N+1} - 2)$ and a is a Geometric Progression that can be simplified to $2^{N+1} - 2$. Hence, the critical point $x = \frac{N2^{N+1}}{2^{N+1}-2} - 1 = \frac{N}{1-2^{-N}} - 1$ where $\frac{N}{1-2^{-N}} < N+1$ and $ceil(x) = N$. The maximum value occurs when only bucket N is nonempty (i.e., all zeros). It is straightforward to show that the critical point is exactly $x = N$. Now, consider any N-bit string, where the Nth bit is 1, if we flip one bit from 1 to zero, the value of the critical point increases. It is straight forward to show that $\frac{b-j2^j}{a-2^j} - \frac{b}{a} > 0$, where j is the index of the flipped bit. □

B Examples of Errors in Approximate Gradient Queue

To better understand the effect of missing elements on the accuracy of the approximate queue, consider the following cases of elements distribution for a maximum priority queue with N buckets:

- Elements are evenly distributed over the queue with frequency $1/\alpha$, which is equivalent to an Exact Gradient Queue with N/α elements,
- $N/2$ elements are present in buckets from 0 to $N/2$ and then a single element is present in bucket indexed $3N/4$, where the concentration of the elements at the beginning of the queue will create an error on the estimation of the index of the maximum element $\varepsilon = ceil(b/a) + u(\alpha) - 3N/4$. We note that in this case $\varepsilon < 0$ because the estimation of $ceil(b/a)$ will be closer to the concentration of elements that is pulling the curvature away from $3N/4$. The error in such cases grows proportional to size of the concentration and inversely proportional to the distance between the low concentration and the high concentration.
- All elements are present, which allows the value $\varepsilon = ceil(b/a) + u(\alpha)$ to be exactly where the maximum element is.