ClubHeap: A High-Speed and Scalable Priority Queue for Programmable Packet Scheduling

Zhikang Chen, *Tsinghua University;* Haoyu Song, *Futurewei Technologies;*
Zhiyu Zhang and Yang Xu, *Fudan University;* Bin Liu, *Tsinghua University*

https://www.usenix.org/conference/nsdi25/presentation/chen-zhikang

This paper is included in the
Proceedings of the 22nd USENIX Symposium on
Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

Open access to the Proceedings of the
22nd USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

ClubHeap: A High-Speed and Scalable Priority Queue for Programmable Packet Scheduling

Zhikang Chen
Tsinghua University

Haoyu Song
Futurewei Technologies

Zhiyu Zhang
Fudan University

Yang Xu
Fudan University

Bin Liu
Tsinghua University

Abstract

While PIFO is a powerful priority queue abstraction to support programmable packet scheduling in network devices, the efficient implementation of PIFO faces multiple challenges in performance and scalability. The existing solutions all fall short of certain requirements. In this paper, we propose ClubHeap to address the problem. On the one hand, we develop a novel hardware-friendly heap data structure to support faster PIFO queue operations that can schedule a flow in every clock cycle, reaching the theoretical lower bound; on the other hand, the optimized hardware architecture reduces the circuit complexity and thus enables a higher clock frequency. The end result is the best scheduling performance in its class. Combined with its inherently better scalability and flexibility, ClubHeap is an ideal solution to be built in programmable switches and SmartNICs to support various scheduling algorithms. We build an FPGA-based hardware prototype and conduct a thorough evaluation by comparing ClubHeap with the other state-of-the-art solutions. ClubHeap also allows graceful trade-offs between throughput and resource consumption through parameter adjustments, making it adaptable on different target devices.

1 Introduction

In a network switch, packet scheduling, which determines the packet delivery order and time, is an essential function in Traffic Manager (TM). To satisfy the diverse application requirements, a number of packet scheduling algorithms have been proposed [6,7,15,21,22,25,31,35,45,63,65], e.g., Weighted Fair Queueing (WFQ) [15], Hierarchical Packet Fair Queueing (HPFQ) [6], Least Slack Time First (LSTF) [31], etc. Fine-grained priority scheduling [3,34,46] based on specific fields, e.g., Shortest Remaining Processing Time (SRPT) [46], is also widely used.

As the programmable network devices play a more important role in Software-Defined Networking (SDN), the programmable schedulers in those devices can support different scheduling algorithms on generic hardware. A popular queue abstraction, Push-In First-Out (PIFO) [51], has been accepted as a foundation to realize a wide range of algorithms. A PIFO is a Priority Queue (PQ) which allows elements to be pushed to an arbitrary location, but can only pop elements from the head. A series of PIFO blocks with mesh interconnections can be configured into a tree to support multi-level hierarchical scheduling. Each PIFO block contains a physical PQ which can be partitioned into multiple logical PQs (named logical PIFOs) at the same level of the tree. In a PIFO block, a flow is inserted in a queue location based on its *rank* [51], the calculated scheduling order or time (e.g., the remaining flow size in SRPT [46]), and popped when it reaches the queue head.

A PIFO queue needs to support two primitives: *push* and *pop*. In a PIFO-based scheduler, a popped flow is immediately pushed back into the queue if it still has unscheduled packets [51], forming a *pop-push* pair. We introduce a *replace* primitive to represent the pair. During scheduling, all operations for a flow are *replace* except for the first *push* and the last *pop*. Therefore, we can use Cycles-Per-Replace (CPR) as a metric to evaluate the throughput performance under the same clock frequency. In pipelinable implementations, we consider the delay between two consecutive *replace* operations as the equivalent CPR because the operations may overlap.

Reducing CPR is an effective means to increase throughput, because the clock frequency is limited in silicon chips. The original PIFO implementation [51] uses shift registers (referred to as SR-PIFO henceforth), in which both *push* and *pop* operations require 2 cycles to complete (i.e., CPR$=4$). Some PIFO queue implementations support parallel *push* and *pop* operations [5,8,26,28,62], enhancing native *replace* support. For example, RPU-BMW [62] allows a *push* every 2 cycles, a *pop* every 3 cycles, but a *replace* every 3 cycles (CPR$=3$) due to its pipeline design. Nevertheless, thus far no scalable PIFO implementation can achieve CPR$=1$ which is the lower limit of a PIFO-based packet scheduler.

In this paper, we propose a novel data structure called

ClubHeap (short for Clustered Binary Heap)[1] to support PIFO queue implementation, and a high-speed and scalable programmable scheduler solution based on it. Our major contributions are summarized as follows.

1. We design ClubHeap, a new PQ data structure supporting the PIFO abstraction. ClubHeap leverages the scalability of the heap-based data structures, and overcomes the challenge of inter-operational data dependencies troubling the other heap-based data structures [8, 26, 28, 62]. ClubHeap is the first heap-based design which supports a fully pipelined implementation with the theoretical lower bound of CPR= 1.

2. To validate the design, we synthesize ClubHeap with a 45nm ASIC toolchain [17, 23, 55] by Design Compiler [53] and implement it on a Xilinx Alveo U280 Data Center Accelerator Card [60]. The prototype supports up to 2^{17} elements in up to 2^8 logical PIFOs, with each element having a priority precision of 32 bits. The prototype achieves a throughput of approximately 200Mpps on FPGA, ensuring 100Gbps line-rate processing in the worst case. We show ClubHeap is the only solution which is scalable in terms of elements, priority levels, and logical PIFOs at the same time.

The rest of the paper is organized as follows. In Sec. 2, we discuss the gap between the requirements of a programmable packet scheduler and the existing hardware implementations using priority queues. In Sec. 3 we introduce the ClubHeap data structure. We present its hardware design in Sec. 4 and its implementation in Sec. 5. A detailed evaluation including comparisons with existing implementations is presented in Sec. 6. We summarize the related works in Sec. 7 and conclude the paper in Sec. 8.

2 Background

2.1 Requirements for PIFO Implementations

The PIFO abstraction can be applied to both programmable switches [10, 50] and SmartNICs [33, 52]. Recent research shows that the priority queues in these two scenarios share the similar requirements as follows [4].

Throughput. Due to the rapid growth of port speed and count on network devices, a single scheduler instance cannot schedule all packets at line rate. For example, a 64x400GE NVIDIA Spectrum SN5400 switch [41] supports up to 33.3 billion packets per second, exceeding any known scheduler capabilities. Therefore, a mesh of PIFO queues is needed for high aggregated scheduling throughput. Fewer queues are needed if each PIFO queue has higher throughput. SmartNICs have a lower peak packet rate than switches, allowing the use of a single PIFO queue. Some recent PIFO queue implementations (e.g., BBQ [4]) can meet the demands of FPGA-based SmartNICs operating for 100GE (148.8Mpps). However, increasing NIC throughput also puts more pressure on the packet scheduler.

Scalability. Scalability is measured in two dimensions: the number of priority levels (P) and the number of elements (N) allowed in the priority queue. In a PIFO queue, the rank of a flow might be a timestamp (e.g., in LSTF [31] or EDF [34]), which often requires high precision. A large P also extends the time span for data retention in the scheduler under non-work-conserving algorithms [21, 29, 57, 67]. On the other hand, today's multi-tenant networks may require the scheduler to handle hundreds of thousands of flows [18, 19, 43], which pose significant scalability challenges for N.

Logical Partitioning. Logical Partitioning (LP) refers to partitioning a physical PIFO block into multiple logical PIFOs. In hierarchical scheduling algorithms, queues at the same level of the scheduling tree can be considered as different logical PIFOs within the same PIFO block. The same applies to the packet queues for different output ports.With LP, a k-port switch requires only $2k$ physical PIFOs, compared to k^2+k PIFOs without LP [4], resulting in a 32.5x reduction for k=64. The significant difference in resource consumption makes LP crucial for the implementation of PIFO-based schedulers.

2.2 Existing Implementations

The requirement for high-performance PQs to support packet scheduling precedes the PIFO abstraction. Numerous PQ implementations exist, but none satisfies all three requirements in Sec. 2.1. We brief three typical implementations below, which can all be used in a PIFO-based scheduler and programmed using Domino [37, 50, 51].

Implementations based on linear data structures. Implementations based on shift registers [9, 13, 42, 51, 54] (e.g., SR-PIFO [51]) and systolic arrays [5, 30, 38] (e.g., OPQ and APQ [5]) fall into this category. For an operation, a shift register performs all comparisons within one cycle, whereas a systolic array breaks it down into multiple sub-operations, and propagates them linearly through the entire data structure in a systolic manner, thereby simplifying the combinational logic for each cycle and improving the clock frequency. APQ, with the highest throughput among them, organizes elements into groups, and leverages the Single-Instruction-Multiple-Data (SIMD) technique to achieve CPR= 1. However, these implementations need $O(N)$ comparators to compare the elements[2] in the PQ in parallel, resulting in significant hardware overhead, and a decrease in clock frequency as N increases. Therefore, these implementations cannot meet the scalability requirements for N. For example, SR-PIFO requires approximately 900K LUTs on an FPGA (∼70% on Xilinx Alveo

[1] Open source available at https://github.com/ClubHeap/ClubHeap.

[2] Each queue element includes a rank and some metadata.

PQ Implementation	Type	CPR	S.P	S.N	LP
SR-PIFO [51]	linear	4	✓	×	✓
OPQ [5]	linear	2	✓	×	×
APQ [5]	linear	1	✓	×	×
BBQ [4]	bucket	2	×	✓	✓
P-Heap [8]	heap	6	✓	✓	×
R-BMW [62]	heap	2	✓	×	×
RPU-BMW [62]	heap	3	✓	✓	×
Pipelined Heap [28]	heap	2	✓	×	✓
H-PQ [26]	hybrid	1∼4	✓	✓	×
ClubHeap	heap	1	✓	✓	✓

Table 1: Comparison of different PQ implementations. (**S.**P and **S.**N represent the scalability for P and N, respectively)

U280 [60]) when $N = 4,096$, and can only operate at a frequency of 40MHz [62]. With CPR= 4, this translates to a throughput of only 10Mpps.

Implementations based on buckets. These implementations assign a bucket to each priority level and group flows with the same priority into the same bucket. BBQ [4] introduces Hierarchical Find-First Set (HFFS) [44, 58] to manage buckets and store elements in SRAMs, achieving scalability with respect to N. However, dedicated hardware for each priority level limits scalability for P. For example, BBQ provides 2^{15} buckets in total [4]. Considering that a PIFO block supports 256 logical PQs (a typical number in [51]), each logical PQ can only support P=2^{15}/256=128, which is unacceptable for many algorithms [6, 22, 31].

Implementations based on heaps. A heap is a tree in which the rank of each node is not greater than that of its child nodes. The operation on a heap-based structure resembles a systolic array but positions sub-operations at distinct tree levels. This design requires only $O(\log N)$ comparators, each with $O(\log\log P)$ latency. P-Heap [8] is the first to exploit the heap structure, achieving scalability for both P and N. BMW-Tree [62] introduces an insertion-balanced multi-way heap with better pipeline support to reduce CPR from 6 to 3. However, both P-Heap and BMW-Tree do not support logical partitioning. Pipelined Heap [28] implements a complete binary heap which can support logical partitioning. However, the global bus required to connect all tree levels becomes a bottleneck in the pipeline, limiting the frequency and making it less scalable with respect to N.

Another challenge for heap-based implementations comes from the inter-operational data dependency problem (detailed in Sec. 2.3), which prevents a fully pipelined implementation with CPR= 1. H-PQ [26] is a hybrid structure of systolic array and a series of heaps, which attempts to eliminate this impact by distributing operations across different heaps. Unfortunately, real traffic may access the same heap continuously, causing the performance instability of the packet scheduler. ClubHeap solves the inter-operational data dependency problem through a novel data structure design.

The characteristics of the aforementioned implementations are summarized in Table 1. We use CPR as the performance indicator for simplicity[3][4]. In summary, implementations based on linear data structures have poor scalability for N, while implementations based on buckets have poor scalability for P. ClubHeap, as a heap-based implementation, achieves a high performance with CPR= 1, presents good scalability for both N and P, and possesses the logical partitioning capability.

2.3 Inter-Operational Data Dependency

Pipelining the queue operation is critical for the scheduling throughput. However, the inter-operational data dependency is a hurdle that prevents the heap-based PIFOs from applying the pipeline technique efficiently. Its solution is the key factor for a high-performance implementation of ClubHeap.

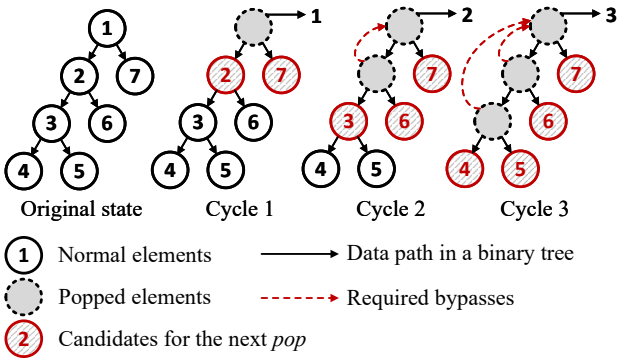


Figure 1: The inter-operational data dependency on a traditional binary heap.

Since it is impossible to predict the rank of the subsequent elements pushed into the PIFO when pushing a current element, the distribution of elements in the heap can be unbalanced. As shown in Fig. 1, in a binary heap, the distribution of the smallest ranked elements may form a dependency chain: 1 is the parent of 2, 2 is the parent of 3, and so on.

Consider consecutive *pop* operations under this circumstance. In Cycle 1, the element 1 at Level 1 (i.e., the root node) is popped, leaving a temporarily empty root, which means both elements stored in its child nodes can be the new minimum element in the heap. Therefore, a comparison must be made between the element 2 and the element 7 at Level 2. In Cycle 2, the second *pop* arrives, which needs to pop the element 2, the smaller one between 2 and 7. However,

[3] Besides CPR, the scheduling throughput is also affected by clock frequency, which may vary for different implementations and different scale specifications (i.e., P and N).

[4] P-Heap claims CPR=2 [8], but it allocates three dependent operations (read, compare, and write) in one cycle. For a fair comparison, we convert one P-Heap cycle to three physical clock cycles as suggested in [28].

the newly arriving instruction just reaches the root node and cannot operate on the element 2 which is still located at Level 2. In order to allow the element 2 to be popped immediately, a bypass from Level 2 to Level 1 (as depicted in the dashed line in Fig. 1) is required. After the element 2 is popped, its two children at Level 3 become the new candidates as well as the element 7. The comparison to elect the next minimum must be made among the three candidates. Similarly, the next *pop* operation arriving in Cycle 3 requires another bypass from Level 3 to Level 1 and increases the number of candidates to four. If the heap has L levels, $L-1$ bypasses are needed from the other levels to Level 1, and a comparison between L candidates needs to be made after $L-1$ consecutive *pop* operations.

In the case of $L = 16$ (i.e., supporting $N = 2^{16} - 1$) and $P = 2^{16}$, the operation to compare 16 16-bit element values and select the minimum one requires a complex combinational circuit, which can severely affect the achievable clock frequency. The wire delay introduced by the bypasses from Level 2, 3, ..., L to Level 1 worsens the problem. In other words, to guarantee a certain performance level, the heap depth is constrained, resulting in poor scalability for N. Because of this, the existing heap-based priority queue implementations [8, 28, 62] are difficult to reconcile with performance and scalability simultaneously.

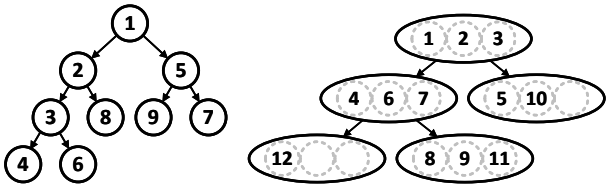
3 ClubHeap

In a traditional heap [8, 28, 62], an operation proceeds along the path from the root to a certain node in the tree structure, resembling a systolic array. Inspired by the SIMD technique where APQ [5] achieves CPR= 1, we organize elements into *clusters* (different from *groups* in APQ, elements in a cluster are rank ordered), so that consecutive operations may only access different elements within a cluster, thereby eliminating the inter-operational data dependency problem.

3.1 Definition and Corollaries

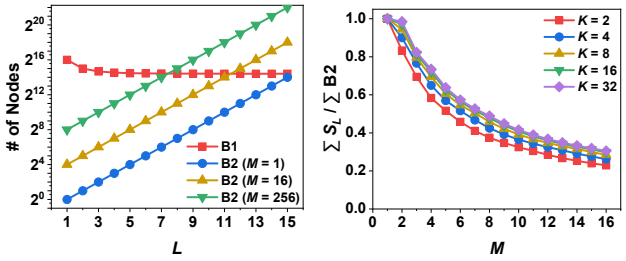
As shown in Fig. 2(a), a traditional binary heap (used in P-Heap [8] and Pipelined Heap [28]) stores a PQ element in each node of the binary tree. Denote the rank of the element on node x as $E(x)$, and consider the rank of an empty node as $+\infty$. Then for any child y of any node x, it holds that $E(x) \le E(y)$.

The key difference between ClubHeap and a traditional binary heap is that each binary tree node in ClubHeap holds more than one element which forms a *cluster*. Each node can store up to K sorted elements. A node that contains K elements is called *full*. For a *non-full* node containing T ($T < K$) elements, the $K-T$ free slots are considered to have the $+\infty$ rank value. We denote the rank array of node x as $E(x)$. For nodes x and y, if $\max(E(x)) \le \min(E(y))$, i.e., the rank of any element in node x is less than or equal to any element in node y, we denote that $E(x) \le E(y)$.



(a) Traditional binary heap. (b) ClubHeap (K = 3).

Figure 2: Comparison of a traditional binary heap and a ClubHeap.



(a) **B1** vs **B2**. (b) S_L vs **B2**.

Figure 3: Comparison of **B1** given by Corollary 2 and **B2** given by the binary tree structure.

Definition. A binary tree is a ClubHeap if and only if: for any child y of a node x, it holds that $E(x) \le E(y)$.

Corollary 1. The definition leads to a corollary: for any child node y of a node x in a ClubHeap, if x is not full, then y must be empty. This is because, for a non-full x, $\max(E(x)) = +\infty$; according to the ClubHeap definition, $\max(E(x)) \le \min(E(y))$, so $\min(E(y)) = +\infty$, meaning that all items in $E(y)$ have the value of $+\infty$ and thus y must be empty.

The corollary implies that the elements in a ClubHeap are always compactly located in the nodes of the upper levels, and the minimum ranked elements must be located at the root node. The corollary below quantitatively describes the concentration of elements in the upper levels of ClubHeap.

Corollary 2. For a ClubHeap containing N elements, the number of non-empty nodes at the L-th level is no more than $\frac{N}{K\left(1-\frac{1}{2^{L-1}}\right)+1}$. The reason is as follows. Assume there are N_L non-empty nodes at the L-th level, so there are at least N_L elements at this level. Meanwhile, due to Corollary 1, there are at least $\frac{1}{2^i} \cdot N_L$ full nodes at the $(L-i)$-th level (i=1,2,...,L-1). Therefore, these nodes contain at least $K \cdot \left(1-\frac{1}{2^{L-1}}\right) \cdot N_L$ elements. Summarizing the two parts of elements, we have $N \ge N_L + K \cdot \left(1-\frac{1}{2^{L-1}}\right) \cdot N_L$, and thus $N_L \le \frac{N}{K\left(1-\frac{1}{2^{L-1}}\right)+1}$.

The concentration of elements is a key factor for ClubHeap to support LP. Consider a ClubHeap forest consisting of M logical PIFOs, with a total of N elements. According to Corol-

lary 2, the total number of nodes at the L-th level across all M logical PIFOs has an upper bound $\frac{N}{K\left(1-\frac{1}{2^{L-1}}\right)+1}$ (**B1**), which may be tighter than the upper bound given by M independent binary trees, i.e., $M \times 2^{L-1}$ (**B2**). Fig. 3(a) presents a comparison of **B1** and **B2** when $K = 2$ and $N = 2^{16} - 2$ (i.e., $L = 15$). It can be observed that **B1** effectively controls the number of nodes at deeper levels.

The memory space required by ClubHeap at the L-th level can be set as $S_L = \min(\mathbf{B1}, \mathbf{B2})$. Fig. 3(b) shows the ratio of the space required by ClubHeap to the space of M physical PIFOs, namely $\sum S_L / \sum \mathbf{B2}$, for different values of K and $N = 2^{16} - K$ (i.e., $L = 16 - \log K$). When $M = 16$, $\sum S_L$ is only 22.9%∼30.5% of $\sum \mathbf{B2}$, indicating the memory efficiency of **B1**. A smaller K leads to a smaller ratio. In summary, **B1** ensures that the memory required by ClubHeap to store nodes does not increase linearly with M, providing better support for LP than BBQ [4].

3.2 Operations

The *push* and *pop* operations on ClubHeap are similar to those on the other heap-based PIFO queues [8, 62]. For the *push* operation, if the root node r is not full, the element is inserted in it directly. Otherwise, the maximum ranked element in r (also including the newly inserted element) is removed and inserted into a child node in the next level. The recursive process continues until the element is settled. For the *pop* operation, the minimum ranked element in the root node r is popped, and then the minimum ranked element among the two children (if non-empty) is raised to fill the node r. The process is recursive at the deeper levels until no more elements are available to promote.

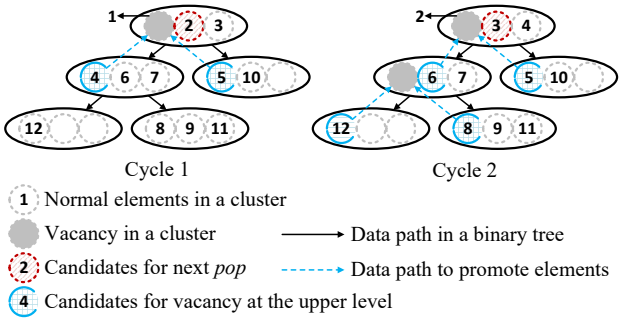


Figure 4: The consecutive *pop* operations on a ClubHeap with no inter-operational data dependency problem.

ClubHeap solves the inter-operational data dependency problem by decomposing the recursion into a sequence of sub-operations without data dependencies. As shown in Fig. 4, we perform consecutive *pop* operations on the ClubHeap in Fig. 2(b). In Cycle 1, element 1 is popped, leaving a vacancy in the root node. Elements 4 and 5 on Level 2 are candidates for promotion. According to ClubHeap's definition, the rank of the promoted element is not less than the remaining elements, so the *pop* operation in Cycle 2 does not depend on the promoted element. In Cycle 2, the element 2 is popped, the element 4 is promoted to Level 1, and elements 12 and 8 on Level 3 compete for the vacancy left on Level 2. Recursively, ClubHeap ensures that each node stores at least $K-1$ smallest ranked elements in its subtree. As long as $K \geq 2$, it can eliminate inter-operational data dependencies between sub-operations on different levels, and ensure that only data paths between adjacent levels are used, thereby reducing pipeline complexity and wire latency.

4 Architectural Design

In the architectural design of ClubHeap, our goal is to achieve CPR= 1 and high clock frequency. We design a pipeline to support CPR= 1, and employ a series of techniques to strive for simplifying the combinational logic circuit in each clock cycle in order to enhance the clock frequency.

In this section, we first present the pipeline architecture in Sec. 4.1 and the technique for efficient cluster storage in Sec. 4.2. We then discuss how the three primitives, i.e., *push*, *pop*, and *replace*, fit into the design in Sec. 4.3. Finally, we discuss the dynamic memory allocation optimization to support the flexible logical partitioning in Sec. 4.4.

4.1 Pipeline Overview

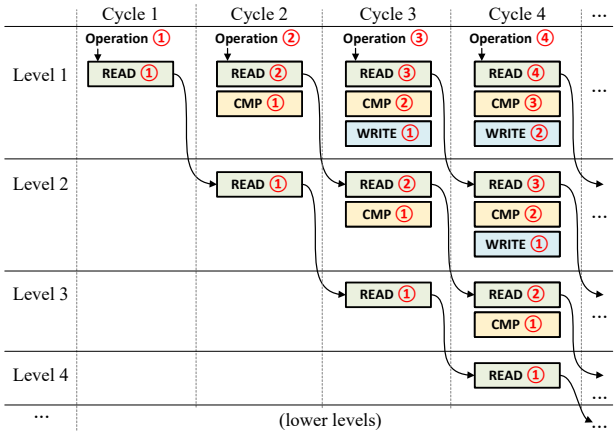


Figure 5: Overview of the ClubHeap pipeline.

In ClubHeap, each primitive, *push*, *pop*, or *replace*, undergoes three stages at each level of the heap: READ, CMP (short for Compare), and WRITE. These stages are executed in different clock cycles to boost clock frequency. Fig. 5 shows the pipeline design of ClubHeap, where four arbitrary primitive operations ①∼④ enters the pipeline sequentially from Cycle 1 to Cycle 4.

When a signal for an operation arrives at a level, it goes through the three stages in sequence. For example, ① arrives at Level 1 in Cycle 1, so its READ stage is performed in Cycle 1, reading the cluster at the root node. In Cycle 2, ① proceeds to the CMP stage at Level 1, comparing the elements within the read cluster. If ① carries an element (*push* or *replace*), it also compares it with the elements in the cluster. In Cycle 3, ① enters the WRITE stage at Level 1, writing the updated cluster back to Level 1 based on CMP results.

For consecutive operations, different stages of different operations overlap. For instance, at Level 1, the CMP stage of ① and the READ stage of ② are performed in the same cycle (Cycle 2), and the WRITE stage of ①, the CMP stage of ②, and the READ stage of ③ are performed in the same cycle (Cycle 3). Each level can handle up to three parallel operations at different stages. Since no two operations can be in the same stage at the same level, each level only requires one hardware module for each stage, eliminating the concern of resource contention.

When a signal for an operation arrives at a level, it continues to propagate downward until it reaches the bottom level. For example, if ① arrives at Level 1 in Cycle 1, it will propagate to Level 2 in Cycle 2 and Level 3 in Cycle 3, and so on. Although in Cycle 3, ① is not yet aware of the comparison results of the CMP stage at Level 2, the heap structure guarantees that the node operated on at Level 3 must be one of the two child nodes of the node operated on at Level 2. Therefore, while the CMP stage is being performed at Level 2, the READ stage can be initiated at Level 3 simultaneously, reading the two child nodes of the current node.

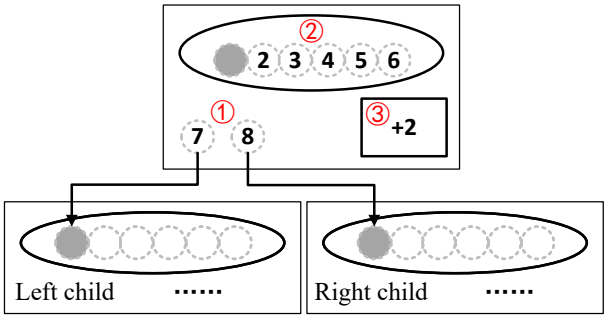
An operation may complete its task at an intermediate level. For example, when ClubHeap is empty, if ① is a *push* primitive, it only needs to insert the element it carries into the root node at Level 1. After this, the operation becomes a no-op (equivalent to pushing a $+\infty$), continuing to propagate downward and perform READ, CMP, and WRITE stages at each level without any real effect.

4.2 Node Data Structure

As explained in Sec. 4.1, because the CMP stage on the previous level has not yet produced the result when the operation enters the READ stage at the current level, both child nodes of the node being processed at the previous level need to be read simultaneously. This implies that the sibling nodes are better to be stored together to enable a single memory access to acquire both nodes. In addition, as shown in Fig. 6, the storage of the clusters adheres to three characteristics.

(1) The minimum ranked element in a cluster for a non-root node is stored in its parent node.

To illustrate the necessity, we assume that all the elements of a cluster are only stored on its corresponding node. According to Corollary 1 in Sec. 3.1, when a *pop* operation is performed on a node x, if x is full and its two child nodes, y and z, are not empty, we must choose the smaller value between $\min(E(y))$ and $\min(E(z))$ and promote the corresponding element to x to fill the vacancy left by the *pop* operation.



① The minimum ranked element in left and right child
② Elements in the cluster (except the minimum ranked)
③ The difference field

Figure 6: The node data structure in ClubHeap.

Assume x is at Level 1, and y and z are at Level 2. Since both elements with $\min(E(y))$ and $\min(E(z))$ are at Level 2, their comparison results can only be obtained after the CMP stage of Level 2 is completed. However, at the moment, the WRITE stage of Level 1 needs to determine the new $E(x)$ based on the comparison result and write it to memory. The data dependency prohibits the pipeline described in Sec. 4.1 from working properly. To fix the problem, it is necessary to store on node x the two elements with the ranks of $\min E(y)$ and $\min E(z)$ in y and z's clusters, so their comparison can be done during the CMP stage of Level 1.

(2) The elements in the cluster at each node are rank ordered.

If the cluster elements are unordered, comparators are needed to determine the minimum ranked element, which is resource consuming and requires complex combinational logic that can affect the achievable clock frequency. With the ordered elements in a cluster, the CMP stage only needs to compare the carried element of the operation with the first element in the cluster to determine the new minimum ranked element.

(3) At each non-leaf node, a field is used to indicate the difference in the number of elements between the two subtrees.

The `difference` field is used by the *push* operation in ClubHeap to achieve insertion-balance of the element distribution, which ensures free location can be found for the new elements before the heap memory is exhausted. Incoming elements are always inserted into the subtree with fewer elements. In case of a tie, the left subtree is chosen. Some existing heap-based implementations [8, 62] use two `counters` to record the number of elements in both subtrees. However, a field only recording the difference is sufficient to achieve the same effect by saving $\log N - 1$ bits per node.

4.3 ClubHeap Processor

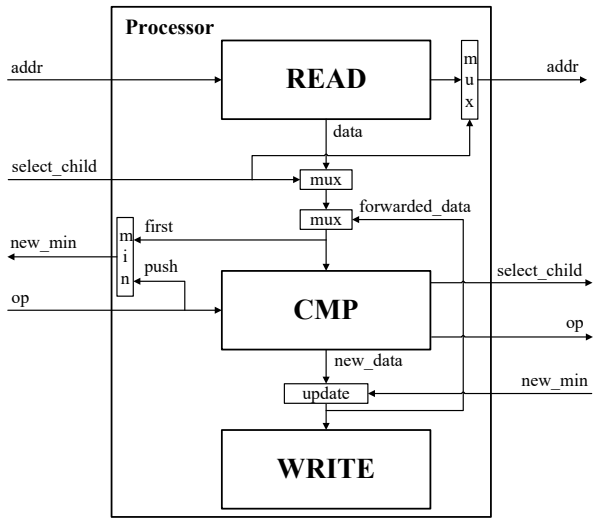


Figure 7: The structure of a processor in ClubHeap.

Each level of ClubHeap has an operation processor and the associated memory. As described in Sec. 4.1, a processor may simultaneously handle the WRITE stage of ①, the CMP stage of ②, and the READ stage of ③. Therefore, a processor in ClubHeap can be divided into three modules with each handling the task for one stage, as shown in Fig. 7. The arrows in the vertical direction represent the data path between the three modules in a single processor. A processor also exchanges data with the upper-level processor and the lower-level processor via the signals in the horizontal direction.

Overall, the READ module reads out a list of rank ordered elements, the CMP module performs parallel comparisons to find the position for writing (*push* or *replace*) or simply removes the first element (*pop*), and the WRITE module writes the result back. The two cycles for the CMP and WRITE stages are similar to the two cycles required for writing into a shift register [51]. Next, we describe the operation of the three modules in detail.

The READ module has only one input signal from the previous level: `addr`, which represents the address line for the nodes to read. As mentioned in Sec. 4.2, sibling nodes are stored together, so the READ module can read a pair of sibling nodes with a single address, and output the data via the `data` signal. Each node contains the data as shown in Fig. 6. Simultaneously, the READ module also obtains the addresses of the respective child nodes of this pair of sibling nodes. The method of storing these addresses is discussed in Sec. 4.4.

The CMP module is the core for executing *push*, *pop*, and *replace* primitives. Firstly, the data from sibling nodes acquired by the READ module passes through a data selector, which selects a branch based on the `select_child` signal generated during the CMP stage of the previous level. Assume the selected node is x. The address of x's child nodes is passed to the READ module of the next level via the `addr` signal, while x's data is sent to the CMP module for comparison. The CMP module determines the current operation type, *push*, *pop*, or *replace*, based on the `op` input signal. It outputs processed node data on the `new_data` signal for writing, and generates a new `op` signal to guide the CMP module at the next level to execute the operation. The `new_data` signal is forwarded as `forwarded_data` when processing the same node in consecutive cycles.

For the *push* operation, we assume the element to be pushed is e. If x is not full, e is directly inserted into the cluster as `new_data` and a new `op` signal, $push(+\infty)$, is generated, which is essentially a no-op. If x is full, e is inserted into the cluster if $rank(e) < \max(E(x))$. In this case, the original element with the $\max(E(x))$ rank value becomes the new e to be pushed to the next level. Otherwise, if $rank(e) \geq \max(E(x))$, the cluster on x is unchanged, and e is to be pushed to the next level. The `difference` field determines the child node for the next level insertion. If $rank(e)$ is less than the minimum rank value of the selected child node (note that the corresponding element e' is stored in node x), e replaces e', and e' is passed to the next level via the output `op` signal; otherwise, the *push* operation with the element e is passed to the next level.

For the *pop* operation, when the CMP module receives the operation from the signal `op`, because the element with $\min(E(x))$ is stored at the previous level, the *pop* operation on the cluster is actually done a cycle earlier. Therefore, the CMP module only needs to select a new minimum to fill the vacancy caused by popping the minimum ranked element, and pass it through `new_min`. Since the elements in a cluster are stored in order, the minimum ranked element can be determined without comparison, i.e., it must be the first element stored at x, or the second smallest ranked element in the cluster of x. Besides, the CMP module compares $\min(E(y))$ and $\min(E(z))$ stored at x, selects the smaller ranked element to be appended to the cluster of x, and passes the *pop* operation via `op` to the corresponding child node at the next level.

For the *replace* operation, since the minimum ranked element is stored at the previous level, the *pop* operation on x is actually performed a cycle earlier in the CMP stage at the previous level, and in the current stage, only two actions are needed: (1) select the new minimum ranked element to fill the vacancy via `new_min`; (2) push a new element e to the cluster, and e can also be selected as `new_min` if its rank value is smaller than that of any other element stored in the node. If $rank(e) < \min(\min(E(y)), \min(E(z)))$, e is directly inserted into the cluster of x, and the new minimum ranked element of x is popped out and returned to the upper level via the `new_min` signal. Otherwise, if $rank(e) \geq \min(\min(E(y)), \min(E(z)))$, the smaller ranked element with the rank value $\min(E(y))$ or $\min(E(z))$ fills the vacancy in the cluster of x, and the $replace(e)$ operation is passed via the `op` signal to the corresponding child node at the next level.

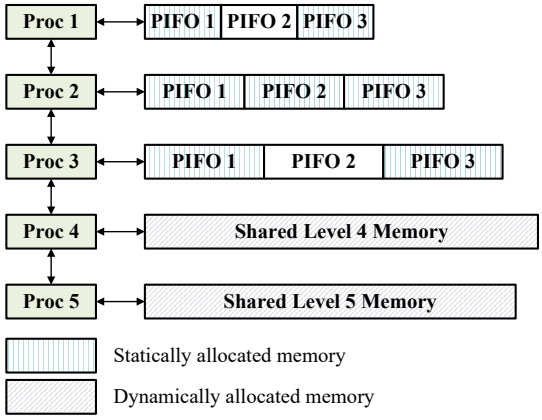


Figure 8: The ClubHeap pipeline structure.

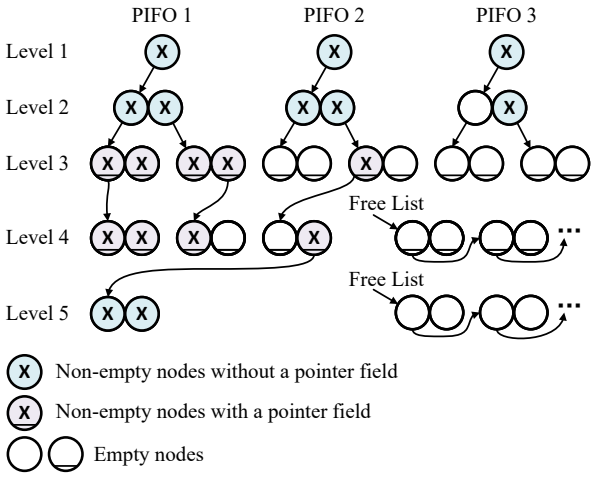


Figure 9: The logical structure of a ClubHeap partitioned into three logical PIFOs.

Finally, the `new_data` output signal from the CMP module is updated by the `new_min` output signal from the data selector in front of the CMP module at the next level, which represents the new minimum ranked element selected by the next level and should be stored at the current level as described in Sec. 4.2. The updated node data is sent to the WRITE module and written to the memory.

4.4 ClubHeap Memory Organization

ClubHeap maintains the memory alongside a processor at each level. The memory is used to support the logical PIFO queues. Since the minimum ranked element of each cluster is actually stored at the upper level, ClubHeap guarantees that the data required in a CMP stage is available at the same level, and thus data dependencies between levels are alleviated. Each processor only accesses the memory at the same level. As shown in Fig. 8, a processor only reads data from memory during the READ stage and writes it back during the WRITE stage, so a dual-port SRAM is sufficient to meet the needs.

Furthermore, as shown in Fig. 8, ClubHeap supports two different forms of memory allocation methods. The static allocation method, similar to P-Heap [8] and BMW-Tree [62], allows for direct child-node address calculation from the current node's address, without the need for storing the child-node's address in memory. Thus, one pointer per node can be saved. However, the drawback is that such allocation implies a complete binary tree, meaning the L-th level of each logical queue requires storage space for 2^{L-1} nodes. With M logical queues provisioned, the memory size required at deeper levels can be prohibitively large.

The discussion in Sec. 3.1 illustrates that the actual storage space required by ClubHeap at the L-th level, S_L, may be significantly less than $\mathbf{B2} = M \times 2^{L-1}$. It is worthwhile to trade a pointer field by reducing **B2** to **B1**. Therefore, as shown in Fig. 8, we employ static memory allocation in shallow levels where **B2**<**B1** and dynamic memory allocation in deep levels where **B1**<**B2** to achieve an optimal space management strategy for different values of M.

As shown in Fig. 9, to achieve dynamic memory allocation for the L-th level, we need to enable the pointer fields of this level and the upper level. During initialization, the pointers at the $(L-1)$-th level are set to *null*, and the pointers at the L-th level are linked into a unidirectional linked list called *free list*. For each pair of sibling nodes in the free list, the left child's pointer field points to the next node, while the right child's pointer is null. If a node is not in the free list, its pointer field either points to its child nodes at the next level if it has non-empty children, or is set to null otherwise. Leaf nodes at the bottom level do not need the pointer field. The `head` address of the free list is maintained by the READ module.

We take dynamic memory allocation at $L = 4$ as an example. The pointer field read from Level 3 is passed as `addr` to Level 4. If `addr` is null (the initial value), the READ module replaces `addr` with `head`, updates `head` with the pointer field of the read node, and resets this pointer field to null. If, at the beginning of the CMP stage at Level 4, it is found that the current node is about to become empty[5] and its sibling node is already empty, a signal is sent to Level 3 to reset the corresponding pointer field to null. Subsequently, during the WRITE stage at Level 4, the pointer field of this empty node is updated to `head`, and the new `head` becomes the address of this empty node. In short, the free list is a stack composed of free nodes, which pops nodes when allocating space and pushes nodes when recycling space. If allocating and recycling occur in the same cycle, the recycled space can be immediately allocated to the new pair of nodes.

[5]The criterion for determining that a node is about to become empty is (1) *push*(+∞) on an empty node, or (2) *pop* on a node with no more than one element. This circuit is only concerned with the valid bits of the elements. The signal to update the pointer field is passed along with `new_min` to the WRITE module of the upper level.

5 Implementation

We implement ClubHeap using 919 lines of Chisel [14] code, and evaluate it on a Xilinx Alveo U280 Data Center Accelerator Card [60] with an FPGA containing 1,303k LUTs, 2,607k Flip-flops (FF), 2,016 Block RAMs (BRAM), and 960 UltraRAMs. The implementation is parameterized, allowing users to specify the parameter K, the maximum number of elements (N), priority levels (P) and logical PIFOs that a ClubHeap can be partitioned into (M). We also implement the existing state-of-the-art PIFO queues BMW-Tree [62] and BBQ [4] on the same FPGA for comparison.

We also analyze the ASIC implementations for some typical specifications with Design Compiler [53] on an open-source 45nm toolchain [17, 23, 40, 55] at a frequency of 800MHz[6]. The reported area is listed in Table 2.

Design	K	N	P	M	Area (mm^2)
BBQ	N/A	2^{17}	2^{16}	1	27.23
ClubHeap	2	2^{17}	2^{16}	1	4.83
ClubHeap	16	2^{17}	2^{16}	1	6.15
ClubHeap	2	2^{17}	2^{32}	2^8	61.00
ClubHeap	16	2^{17}	2^{32}	2^8	63.60

Table 2: ASIC area comparison.

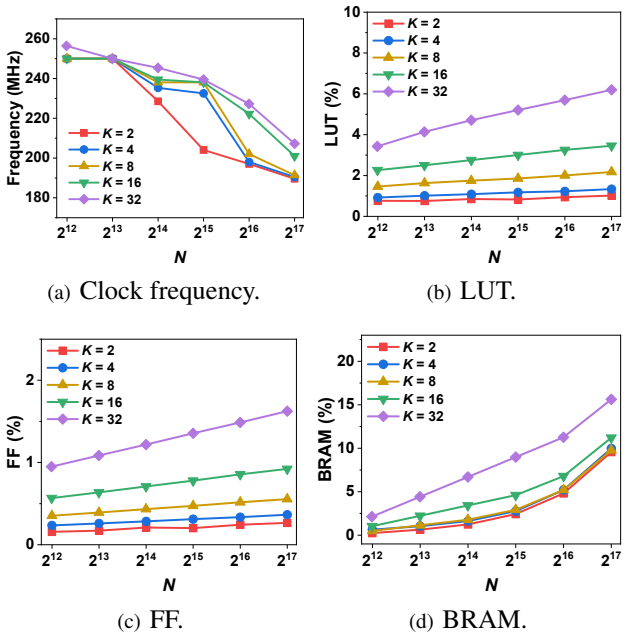
Table 2 indicates ClubHeap consumes only 17.7% (K=2) or 22.6% (K=16) of area used by BBQ for the same specification[7]. BMW-Tree is not listed here because the open-source 4-way implementation can not support N=2^{17}. We conduct a more detailed comparison between ClubHeap, BMW-Tree, and BBQ on FPGA in Sec. 6.

6 Evaluation

We evaluate ClubHeap through experiments and analysis to show that ClubHeap can meet the demands mentioned in Sec. 2.1, and is the most performant PQ implementation for a PIFO scheduler so far. We first introduce the experimental methodology in Sec. 6.1, then discuss the selection of parameter K in Sec. 6.2, and compare ClubHeap with the state-of-the-art implementations, BMW-Tree and BBQ, in Sec. 6.3. In Sec. 6.4, we show that a clustered multi-way tree is inferior in terms of throughput and resource consumption, which explains why we choose a binary heap rather than a multi-way one.

[6]BBQ claims their *logic* can achieve 3.1GHz in a 7nm process [4, 59], but the bottleneck in large SRAM access is ignored.

[7]Since a ClubHeap with L levels can contain a maximum of $N = K(2^L - 1) = 2^{K+L} - K$ elements, there may be slight variations in N for different values of K. For example, when K=2 and L=15, N=65,534, while for K=4 and L=14, N=65,532. Since these minor differences have almost negligible practical effects, both cases are denoted as $N = 2^{16}$ in our evaluation and considered as the same data point.



(a) Clock frequency. (b) LUT. (c) FF. (d) BRAM.

Figure 10: Frequency and resource consumption of ClubHeap with different K.

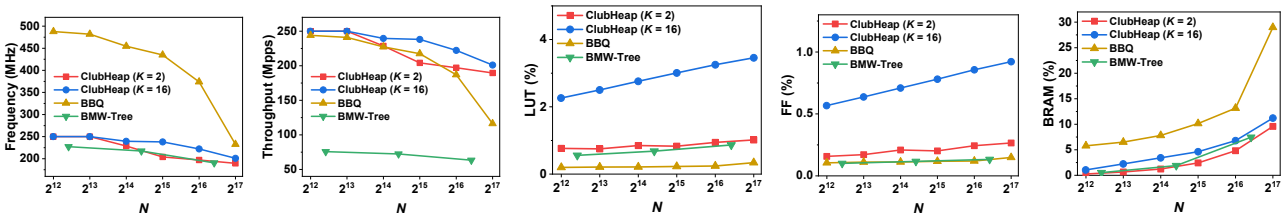
6.1 Methodology

We evaluate the FPGA implementations of ClubHeap, BMW-Tree, and BBQ for comparison. Clock frequency and resource consumption are reported by Xilinx Vivado, which may differ from existing papers [4, 62] due to different FPGAs. We measure the achievable clock frequency by testing different cycles (step size: 0.025ns) and selecting the minimum allowed. The throughput is obtained via FPGA simulation, as it is independent of traffic characteristics or scheduling algorithms for these structures. For resource consumption, we focus on LUTs, FFs, and BRAMs; UltraRAMs are converted to BRAMs with equivalent storage capacity in results.

Unless otherwise specified, we assume each scheduler element includes a 16-bit rank value (P=2^{16}) and 32-bit metadata (for flow ID, packet ID, etc.), sufficient for most PIFO-based scheduling algorithms [51, 62]. For a fair comparison, we focus on scenarios with one logical PIFO (M=1), as ClubHeap better supports logical partitioning than BMW-Tree and BBQ.

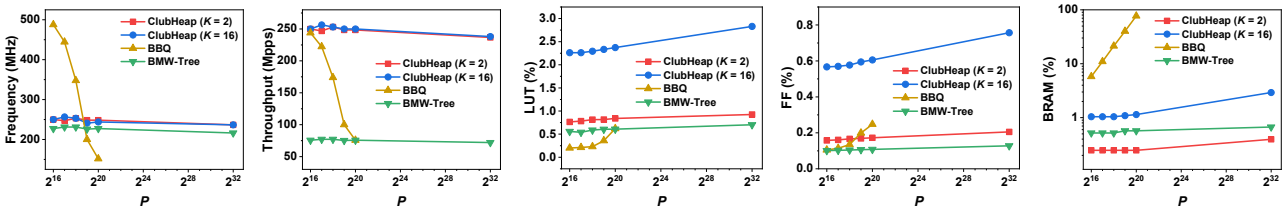
6.2 Effects of Different K Values

We first evaluate the clock frequency and resource consumption of ClubHeap with different K values. As shown in Fig. 10(a), when N=2^{12} and N=2^{13}, clock frequencies are approximately 250MHz regardless of K. As N increases, the clock frequency decreases, and differences in clock frequency begin to emerge for different K. Experiment results indicate that for the same N value, a larger K generally corresponds to a higher clock frequency. At N=2^{17}, the frequency for K=32 is



(a) Clock frequency. (b) Throughput. (c) LUT. (d) FF. (e) BRAM.

Figure 11: Frequency, throughput and resource consumption for ClubHeap, BBQ and BMW-Tree at different N.



(a) Clock frequency. (b) Throughput. (c) LUT. (d) FF. (e) BRAM.

Figure 12: Frequency, throughput, and resource consumption for ClubHeap, BBQ, and BMW-Tree at different P.

207.25MHz, which is approximately 10% higher than that for K=2 (189.57MHz). This is because a larger K results in fewer levels in ClubHeap, reducing the place and route complexity.

However, a larger K requires more resources on the FPGA, as shown in Fig. 10(b), 10(c), and 10(d). At N=2^{17}, a ClubHeap with K=32 requires relatively 6.2% LUTs and 1.6% FFs, which are approximately 6x those for K=2. Additionally, the growth rate of LUT and FF overhead with increasing N is also higher for a larger K. Comparing N=2^{12} and N=2^{17}, for a 32x increase in N, ClubHeap with K=2 experiences a 33% increase in LUTs and a 68% increase in FFs, while ClubHeap with K=32 experiences an 80% increase in LUTs and a 71% increase in FFs. Overall, ClubHeap's demand for LUTs and FFs is approximately $O(\log N)$, ensuring its good scalability.

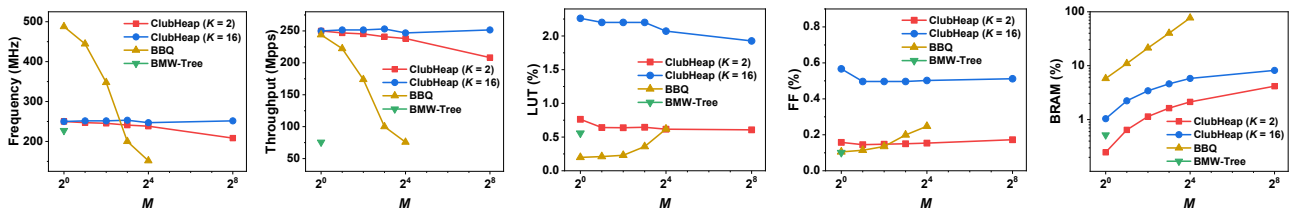
For BRAM, although the same number of elements requires the same space, a larger K implies more wasted space and thus requires more BRAMs. Since sibling nodes are stored at the same address, there are only 2^{L-2} addresses for the 2^{L-1} nodes on the L-th level. When L is small, only a small portion of a BRAM block is utilized, leaving the unused space wasted. At N=2^{12}, the number of BRAM blocks required by ClubHeap with K=32 is 8.6x that for K=2. At deeper levels, a BRAM block can be fully utilized, and the number of BRAM blocks is mainly determined by the number of storage bits. A smaller K indicates each node stores fewer elements, leading to more nodes and a greater demand for the difference field and the pointer for dynamic memory allocation. Therefore for a larger N, a smaller K results in more space requirements, reducing the relative savings compared to a larger K. At N=2^{17}, the BRAM consumption for K=32 is only 1.63x that for K=2.

In summary, the analysis indicates that for the same N, ClubHeap with a larger K exhibits a higher clock frequency but also demands more resources. When applying ClubHeap in practice, K can be selected based on the specific requirements as a tradeoff between performance and resource consumption. A proper K value could be 2, 4, 8, or 16.

6.3 Comparison with BMW-Tree and BBQ

We compare ClubHeap with K=2 and K=16 to RPU-BMW, which is the scalable version of BMW-Tree [62], and BBQ [4]. The comparison focuses on three parameters: the number of elements (N), the number of priority levels (P) and the number of logical PIFOs (M).

Evaluation on N. In Fig. 11 we fix P=2^{16} and M=1 to evaluate the scalability of the implementations on N. We observe that for all three PQ implementations, the critical path affecting clock frequency is the combinational logic when N is small, whereas it shifts to SRAM access when N is large, because they all access SRAM in one clock cycle as part of the pipeline design. ClubHeap and BMW-Tree have similar comparator-dominated combinational logic latency; thus, they have similar frequencies. ClubHeap achieves a throughput 3x higher than BMW-Tree with CPR= 3. On the other hand, BBQ's HFFS uses the least combinational logic and has the highest frequency for small N, but the frequency drops rapidly as N increases due to the memory access constraints. As shown in Fig. 11(a), when N=2^{17}, BBQ's clock frequency is comparable to ClubHeap and BMW-Tree, making CPR a more decisive factor for throughput. Fig. 11(b) indicates that ClubHeap's throughput is approximately equal to BBQ for small N, but 63% (K=2) or 72% (K=16) higher when N=2^{17},



(a) Clock frequency. (b) Throughput. (c) LUT. (d) FF. (e) BRAM.

Figure 13: Frequency, throughput, and resource consumption for ClubHeap, BBQ, and BMW-Tree at different M.

showing better scalability for N than BBQ.

As shown in Fig. 11(c) and 11(d), ClubHeap processors at each level require more LUT and FF resources than BMW-Tree and BBQ. As shown in Fig. 11(e), ClubHeap consumes the similar BRAMs as BMW-Tree, but much fewer than BBQ. At N=2^{17}, ClubHeap consumes 33% (K=2) or 39% (K=16) fewer BRAMs than BBQ. Note that the proportion of LUTs and FFs used by these structures to the total FPGA resources is significantly less than that of BRAMs. Therefore, it is worthwhile for ClubHeap to consume more LUTs and FFs to save BRAMs compared to BBQ.

The 45nm ASIC analysis in Sec. 5 supports the same conclusion. For N=2^{17}, P=2^{16}, and M=1, the clock frequency bottleneck for both ClubHeap and BBQ is SRAM access. BBQ requires larger SRAM which causes higher access latency, while ClubHeap's memory is distributed across levels which has lower access latency. Additionally, ClubHeap consumes less chip area than BBQ. The following evaluations indicate that ClubHeap's advantages become more pronounced as P and M increase. We believe that ClubHeap will remain a better choice at more advanced process nodes.

Evaluation on P. We evaluate the implementations for different P with N=2^{12} and M=1 in Fig. 12 (for BMW-Tree, we select the nearest N=5,460). When the range of ranks is 2x wider, the only changes needed in ClubHeap are an additional bit for the rank field and an additional 1-bit width of the comparators.When P increases from 2^{16} to 2^{20}, the clock frequency of ClubHeap only drops less than 3%. As a heap-based PQ implementation, BMW-Tree exhibits similar scalability for P. However, BBQ needs to expand the number of buckets to support a larger range of P, resulting in poorer scalability. As shown in Fig. 12(a), the large number of buckets poses challenges in routing, causing its clock frequency to drop lower than that of ClubHeap when P=2^{19}. Considering CPR, ClubHeap achieves a throughput 3.28x higher than BBQ when P=2^{20} as shown in Fig. 12(b). BBQ with a larger P fails to be synthesized on our FPGA because of the exhausted BRAMs, while ClubHeap can scale up to P=2^{32} with a cost of only 5.5% frequency reduction.

As shown in Fig. 12(c), 12(d), and 12(e), with the increase of P, BBQ requires more levels or a larger bitmap width for HFFS, resulting in higher consumption of LUTs, FFs, and BRAMs following an $O(M)$ trend. Among them, BRAM consumption grows the fastest, reaching 70x more than ClubHeap (K=2) when P=2^{20}. In contrast, the LUT and FF consumption for ClubHeap grows slowly. When P increases from 2^{16} to 2^{20}, ClubHeap consumes only about 10% additional LUTs and FFs, and the BRAM consumption remains constant when the additional bits do not cause the node size to exceed the width provided by BRAMs at P=2^{16}.

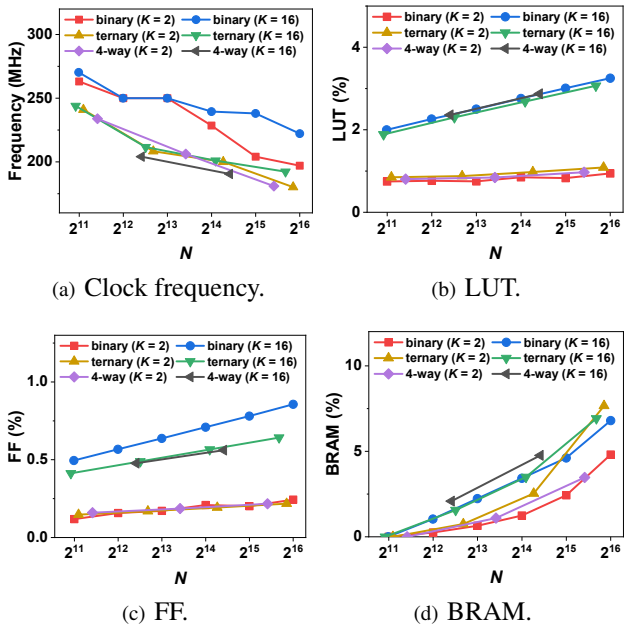
Evaluation on M. Support for logical partitioning is an advantage of ClubHeap over BMW-Tree, so BMW-Tree is represented by only one point (M=1) in Fig. 13. BBQ supports logical partitioning, but it requires evenly dividing buckets among various logical PIFOs. With a constant N and P, M logical PIFOs in BBQ use M times more buckets, which is equivalent to a M times larger P. Fig. 13 shows the frequency, throughput, and resource consumption when a physical queue with N=2^{12} and P=2^{16} is divided into M logical PIFOs. We draw a similar conclusion as for Fig. 12. The frequency of ClubHeap with K=2 decreases by only 16.7% when M scales to 2^8, while at K=16 there is no frequency reduction.

As shown in Fig. 13(c) and 13(d), the LUT and FF consumption decreases when M gets larger, because more memory is synthesized as BRAMs rather than LUTMEMs. It is worth mentioning that the memory space required for ClubHeap has an upper limit independent of M, bounded by **B1**.

In summary, although BBQ achieves a high throughput for small N, it lacks scalability on P and M. BMW-Tree, on the other hand, offers scalability on both N and P, but has lower throughput and does not support logical partitioning.

6.4 Binary Heap vs Multi-way Heap

We compare ClubHeap with clustered ternary and 4-way heaps to explain the rationale for choosing a binary heap rather than a multi-way one. As shown in Fig. 14(a), the frequency of a clustered multi-way heap is always lower than that of ClubHeap. The frequency degradation is because the clustered ternary heap requires more comparisons and selections during the process of selecting `new_min`, resulting in a longer critical path. Meanwhile, as shown in Fig. 14(b), 14(c), and 14(d), the clustered multi-way heap does not reduce the resource consumption as well.



(a) Clock frequency. (b) LUT.

(c) FF. (d) BRAM.

Figure 14: Frequency and resource consumption comparison of ClubHeap and a clustered ternary / 4-way heap.

7 Related Works

PIFO [51] is a classical programmable packet scheduling abstraction that can express a wide range of algorithms, but it faces two limitations. (1) PIFO's programmability is achieved through a mesh composed of multiple PQs, but its original implementation , SR-PIFO [51], encounters significant challenges in terms of throughput and scalability. (2) PIFO assumes that the rank of a packet is already calculated when it enters the scheduler and is unchanged thereafter. The assumption makes the abstraction unable to express algorithms with dynamic ranks, e.g. pFabric [3], which introduces a starvation avoidance mechanism to SRPT [46].

Optimization of PIFO. Replacing SR-PIFO with other PQs can help improve the throughput and scalability of PIFO. OPQ and APQ [5] introduce the Single-Instruction-Multiple-Data (SIMD) technique on systolic arrays to enhance throughput, but they lack scalability on the number of elements. Calendar Queues [11], CPI [58], and BBQ [4] implement PQs using buckets, but the number of buckets limits the number of priorities. P-Heap [8], Pipelined Heap [28], H-PQ [26], and BMW-Tree [62, 68] distribute the storage and computation of a PQ across levels using heaps, achieving good scalability. However, the inter-operational data dependency problem discussed in Sec. 2.3 restricts their performance. ClubHeap, proposed in this paper, is the first high-speed and scalable PIFO queue implementation, which can help PIFO overcome the first limitation.

Approximation of PIFO. Another direction to overcome the first limitation is to approximate PIFO by eliminating the need for using a mesh of PQs. UPS [36] employs LSTF as an approximate universal packet scheduling algorithm, enabling programmable packet scheduling with a single PQ. SP-PIFO [1] utilizes a group of FIFOs, clustering flows with similar ranks into the same FIFO to achieve scheduling approximation. FDPA [12], AFQ [47], PCQ [48], PUPD [56], GearBox [20], QCluster [61], and PR-AQM [32] also use multiple FIFOs for approximate scheduling. AIFO [64], on the other hand, only uses a single FIFO, which limits its ability to approximate scheduling order but ensures approximation for whether a packet is admitted or dropped. FAIFO [69] extends AIFO by considering information freshness to better support time-sensitive applications. RIFO [39] improves AIFO's admission control using min-max linear normalization. PACKS [2] is a hybrid structure of SP-PIFO and AIFO, which achieves approximation of PIFO in terms of both scheduling and admission.

Extension of PIFO. New abstractions are proposed to overcome the second limitation. PIEO [49] allows the PQs to pop elements from any location rather than the head only, enabling scheduling with eligibility predicates. PIPO [66] implements an approximated PIEO with multiple PIFO queues for bounded-delay scheduling algorithms. PCSQ [27] extends PIPO to large-scale deterministic networks with long-distance links. CIPO [24] proposes an approximation of PIEO with two-dimensional predicates. DR-PIFO [16] combines a PIFO queue with a group of FIFOs to support dynamic ranks. Eiffel [44] introduces a queue structure based on Circular FFS to provide on-dequeue scheduling. ClubHeap does not support these extended abstractions, but it can be incorporated as a PIFO queue in their implementations.

8 Conclusion

ClubHeap is a novel heap-based PIFO queue implementation. It overcomes the performance and scalability challenges of PIFO's implementations to achieve line-rate scheduling for a large number of flows with a wide range of priority levels. It is beneficial to any programmable schedulers implemented with the PIFO abstraction on switches or SmartNICs based on either FPGA or ASIC.

Acknowledgement. We thank the anonymous reviewers and the paper shepherd Stefan Schmid for their insightful comments and suggestions, which help improve this paper. We thank Chuwen Zhang (Tsinghua University), Yongzheng Zhang (ShanghaiTech University), Yinuo Jia (Wuhan University), and Haodong Wang (Northwestern Polytechnical University) for helping in conducting experiments. This work is supported by National Key Research and Development Program of China (2024YFE0203900), NSFC (62032013, 62272258, 62172108), NSFC-RGC (62061160489). The corresponding author is Bin Liu (lmyujie@gmail.com).

References

[1] 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, 2020. USENIX Association.

[2] Albert Gran Alcoz, Balazs Vass, Pooria Namyar, Behnaz Arzani, Gabor Retvari, and Laurent Vanbever. Everything matters in programmable packet scheduling. In *NSDI*. USENIX, 2025.

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: minimal near-optimal datacenter transport. *SIGCOMM Comput. Commun. Rev.*, 43(4):435–446, 2013.

[4] Nirav Atre, Hugo Sadok, and Justine Sherry. BBQ: A fast and scalable integer priority queue for hardware packet scheduling. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 455–475, Santa Clara, CA, 2024. USENIX Association.

[5] Imad Benacer, François-Raymond Boyer, and Yvon Savaria. A fast, single-instruction–multiple-data, scalable priority queue. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(10):1939–1952, 2018.

[6] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. *SIGCOMM Comput. Commun. Rev.*, 26(4):143–156, 1996.

[7] Jon C. R. Bennett and Hui Zhang. WF2Q: worst-case fair weighted fair queueing. In *Proceedings of the Fifteenth Annual Joint Conference of the IEEE Computer and Communications Societies Conference on The Conference on Computer Communications (INFOCOM) - Volume 1*, INFOCOM'96, page 120–128, USA, 1996. IEEE Computer Society.

[8] Ranjita Bhagwan and Bill 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 Societies*, volume 2, pages 538–547, 2000.

[9] Gedare Bloom, Gabriel Parmer, Bhagirath Narahari, and Rahul Simha. Shared hardware data structures for hard real-time systems. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, page 133–142, New York, NY, USA, 2012. Association for Computing Machinery.

[10] 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. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, 2013.

[11] Randy Brown. Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[12] Carmelo Cascone, Nicola Bonelli, Luca Bianchi, Antonio Capone, and Brunilde Sansò. Towards approximate fair bandwidth sharing via dynamic priority queuing. In *2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–6, 2017.

[13] Ravikesh Chandra and Oliver Sinnen. Improving application performance with hardware data structures. In *2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–4, 2010.

[14] ChipsAlliance. Chisel. https://www.chisel-lang.org/, 2024.

[15] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1–12, 1989.

[16] Mostafa Elbediwy, Bill Pontikakis, Alireza Ghaffari, Jean-Pierre David, and Yvon Savaria. DR-PIFO: A dynamic ranking packet scheduler using a push-in-first-out queue. *IEEE Transactions on Network and Service Management*, 21(1):355–371, 2024.

[17] Hewlett Packard Enterprise. CACTI. https://github.com/HewlettPackard/cacti, 2017.

[18] 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, 2017. USENIX Association.

[19] 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, 2018. USENIX Association.

[20] 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, 2022. USENIX Association.

[21] S. Jamaloddin Golestani. A stop-and-go queueing framework for congestion management. *SIGCOMM Comput. Commun. Rev.*, 20(4):8–18, 1990.

[22] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking*, 5(5):690–704, 1997.

[23] Bespoke Silicon Group. BSG Block-box SRAM Generator. https://github.com/bespoke-silicon-group/bsg_fakeram, 2022.

[24] Feng Guo, Shidong Sun, Junjie Hu, Ning Zhang, and Zhiqiang Lv. CIPO: Efficient, lightweight and programmable packet scheduling. *Computer Networks*, 245(C), 2024.

[25] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. Multi-Queue fair queuing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 301–314, Renton, WA, 2019. USENIX Association.

[26] Muhuan Huang, Kevin Lim, and Jason Cong. A scalable, high-performance customized priority queue. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2014.

[27] Yudong Huang, Shuo Wang, Shiyin Zhu, Guoyu Peng, Xinyuan Zhang, Tian Pan, Tao Huang, Lei Zhou, Zuopin Cheng, Daorong Guo, Hui Lin, Lianqing Zhang, Juyan Lei, Liangzhang Xu, Wei Wang, Xinmin Liu, Xuejun You, and Yunjie Liu. Poster: Programmable cycle-specified queue for deterministic networking. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 1132–1134, New York, NY, USA, 2023. Association for Computing Machinery.

[28] 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, 2007.

[29] C.R. Kalmanek, H. Kanakia, and S. Keshav. Rate controlled servers for very high-speed networks. In *[Proceedings] GLOBECOM '90: IEEE Global Telecommunications Conference and Exhibition*, pages 12–20 vol.1, 1990.

[30] Pierre Lavoie, David Haccoun, and Yvon Savaria. A systolic architecture for fast stack sequential decoders. *IEEE Transactions on Communications*, 42(234):324–335, 1994.

[31] Joseph Y. T. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4(1–4):209–219, 1989.

[32] Ziyong Li, Yuxiang Hu, Le Tian, and Zhao Lv. Packet rank-aware active queue management for programmable flow scheduling. *Computer Networks*, 225:109632, 2023.

[33] 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, 2020.

[34] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[35] Paul E. McKenney. Stochastic fairness queueing. In *Proceedings. IEEE INFOCOM '90: Ninth Annual Joint Conference of the IEEE Computer and Communications*, pages 733–740 vol.2, 1990.

[36] 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, Santa Clara, CA, 2016. USENIX Association.

[37] Anshuman Mohan, Yunhe Liu, Nate Foster, Tobias Kappé, and Dexter Kozen. Formal abstractions for packet scheduling. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

[38] Sung-Whan Moon, Kang G. Shin, and Jennifer Rexford. Scalable hardware priority queue architectures for high-speed packet switches. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium (RTTAS)*, pages 203–212, 1997.

[39] Habib Mostafaei, Maciej Pacut, and Stefan Schmid. RIFO: Pushing the efficiency of programmable packet schedulers, 2024.

[40] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 3–14, 2007.

[41] NVIDIA. NVIDIA Spectrum SN5000 Series Switches. https://www.nvidia.com/en-us/networking/ethernet-switching/, 2023.

[42] Dan Picker and Ronald D. Fellman. A VLSI priority packet queue with inheritance and overwrite. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(2):245–253, 1995.

[43] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, SIGCOMM '17, page 404–417, New York, NY, USA, 2017. Association for Computing Machinery.

[44] 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, 2019. USENIX Association.

[45] Hanrijanto Sariowan, Rene L. Cruz, and George C. Polyzos. SCED: a generalized scheduling policy for guaranteeing quality-of-service. *IEEE/ACM Transactions on Networking*, 7(5):669–684, 1999.

[46] Linus E. Schrage and Louis W. Miller. The queue M/G/1 with the shortest remaining processing time discipline. *Operations Research*, 14(4):670–684, 1966.

[47] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 1–16, USA, 2018. USENIX Association.

[48] 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 *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 685–700, USA, 2020. USENIX Association.

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

[50] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 15–28, New York, NY, USA, 2016. Association for Computing Machinery.

[51] 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, page 44–57, New York, NY, USA, 2016. Association for Computing Machinery.

[52] Brent Stephens, Aditya Akella, and Michael 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, 2019. USENIX Association.

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

[54] Kenji Toda, Kenji Nishida, Eiichi Takahashi, Nick Michell, and Yoshinori Yamaguchi. Implementation of a priority forwarding router chip for real-time interconnection networks. In *Second Workshop on Parallel and Distributed Real-Time Systems*, pages 166–175, 1994.

[55] NC State University. NCSU FreePDK45. https://eda.ncsu.edu/freepdk/freepdk45/, 2023.

[56] Balázs Vass, Csaba Sarkadi, and Gábor Rétvári. Programmable packet scheduling with sp-pifo: Theory, algorithms and evaluation. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6, 2022.

[57] D.C. Verma, H. Zhang, and D. Ferrari. Delay jitter control for real-time communication in a packet switching network. In *Proceedings of TRICOMM '91: IEEE Conference on Communications Software: Communications for Distributed Applications and Systems*, pages 35–43, 1991.

[58] 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.

[59] 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.

[60] Xilinx. Alveo U280 Data Center Accelerator. https://www.avnet.com/opasdata/d120001/medias/docus/196/XLX-A-U280-A32G-DEV-G-Datasheet.pdf, 2019.

[61] Tong Yang, Jizhou Li, Yikai Zhao, Kaicheng Yang, Hao Wang, Jie Jiang, Yinda Zhang, and Nicholas Zhang. Qcluster: Clustering packets for flow scheduling. In *Proceedings of the ACM Web Conference 2022*, WWW '22, page 1752–1763, New York, NY, USA, 2022. Association for Computing Machinery.

[62] 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*, ACM SIGCOMM '23, page 208–219, New York, NY, USA, 2023. Association for Computing Machinery.

[63] Xin Yu, Wei Chen, and Ye Tian. OWFQ: Reducing packet drops for approximate weighted fair queueing with calendar queues. In *2023 9th International Conference on Computer and Communications (ICCC)*, pages 540–544, 2023.

[64] 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, page 179–193, New York, NY, USA, 2021. Association for Computing Machinery.

[65] Zhuolong Yu, Jingfeng Wu, Vladimir Braverman, Ion Stoica, and Xin Jin. Twenty years after: Hierarchical Core-Stateless fair queueing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 29–45. USENIX Association, 2021.

[66] Chuwen Zhang, Zhikang Chen, Haoyu Song, Ruyi Yao, Yang Xu, Yi Wang, Ji Miao, and Bin Liu. PIPO: Efficient programmable scheduling for time sensitive networking. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–11, 2021.

[67] Hui Zhang and Domenico Ferrari. Rate-controlled service disciplines. *J. High Speed Netw.*, 3(4):389–412, October 1994.

[68] Zhiyu Zhang, Shili Chen, Ruyi Yao, Ruoshi Sun, Hao Mei, Hao Wang, Zixuan Chen, Gaojian Fang, Yibo Fan, Wanxin Shi, Sen Liu, and Yang Xu. vpifo: Virtualized packet scheduler for programmable hierarchical scheduling in high-speed networks. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 983–999, New York, NY, USA, 2024. Association for Computing Machinery.

[69] Meng-yuan Zhu, Ke-fan Chen, Zhuo Chen, and Na Lv. FAIFO: UAV-assisted IoT programmable packet scheduling considering freshness. *Ad Hoc Networks*, 134(C), 2022.