BMW Tree: Large-scale, High-throughput and Modular PIFO Implementation using Balanced Multi-Way Sorting Tree

Ruyi Yao
Fudan University

Zhiyu Zhang
Fudan University

Gaojian Fang
Fudan University

Peixuan Gao
New York University

Sen Liu
Fudan University

Yibo Fan
Fudan University

Yang Xu*
Fudan University,
Peng Cheng Laboratory

H. Jonathan Chao
New York University

ABSTRACT

Push-In-First-Out (PIFO) queue has been extensively studied as a programmable scheduler. To achieve accurate, large-scale, and high-throughput PIFO implementation, we propose the Balanced Multi-way (BMW) Sorting Tree for real-time packet sorting. The tree is highly modularized, insertion-balanced and pipeline-friendly with autonomous nodes.

Based on it, we design two simple and efficient hardware designs. The first one is a register-based (R-BMW) scheme. With a pipeline, it features an impressively high and stable throughput without any frequency reduction theoretically even under more levels. We then propose Ranking Processing Units to drive the BMW-Tree (RPU-BMW) to improve the scalability, where nodes are stored in SRAMs and dynamically loaded into/off from RPUs. As the capacity of BMW-Tree grows exponentially, only a few RPUs are needed for a large scale.

The evaluation shows that when deployed on the Xilinx Alveo U200 card, R-BMW improves the throughput by 4.8x compared to the original PIFO implementation, while exhibiting a similar capacity. RPU-BMW is synthesized in GlobalFoundries 28nm process, costing a modest 0.522% (1.043mm^2) chip area and 0.57MB off-chip memory to support 87k flows at 200Mpps. To our best knowledge, RPU-BMW is the first accurate PIFO implementation supporting over 80k flows at as fast as 200Mpps.

CCS CONCEPTS

• **Networks** → **Programmable networks**; • **Hardware** → **Networking hardware**;

*Corresponding Author

ACM SIGCOMM '23, September 10, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3604862

KEYWORDS

Programmable Packet Scheduler; Networking Hardware

ACM Reference Format:
Ruyi Yao, Zhiyu Zhang, Gaojian Fang, Peixuan Gao, Sen Liu, Yibo Fan, Yang Xu, and H. Jonathan Chao. 2023. BMW Tree: Large-scale, High-throughput and Modular PIFO Implementation using Balanced Multi-Way Sorting Tree. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10, 2023, New York, NY, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3603269.3604862

1 INTRODUCTION

In the traffic manager (TM), the scheduler is such an indispensable component that it guarantees the Quality of Service (QoS) and Quality of Experience (QoE) required by a Service Level Agreement (SLA). Since the last century, academia and industry have proposed numerous scheduling algorithms to achieve various application requirements [1–6]. With the development of technologies, there are still many new scenarios calling for appropriate scheduling algorithms, such as distributed machine learning and high performance computing. However, due to the high cost of designing and deploying switch ASICs, seldom scheduling algorithms are put into production.

With the rise of programmable data planes, programmable schedulers are under the spotlight and receiving more and more attention. A programmable scheduler is such a magic bullet that it allows most existing or brand-new scheduling algorithms to be implemented with generic hardware, which enables rapid testing and deployment. In addition to being expressive enough, a programmable scheduler should have high throughput and a large scale to work in next-generation networks.

Programmable schedulers should be fast to keep pace with the line rate to avoid lagging, which is typically 100-400 Gbps [7]. It is such a big challenge that a medium-sized packet has to be processed and forwarded within 10.24 ns when the line rate is 400 Gbps. Only a few scheduling algorithms have been reported to work in line-rate switches: DRR, traffic shaping, and strict priorities [8]. A fast enough programmable scheduler should break the awkward situation and make various algorithms more feasible.

Programmable schedulers are expected to be scalable to support enormous numbers of flows. In a modern multi-tenant data center, it is not uncommon for hundreds of virtual machines or containers to run on a single physical machine, with each generating tens of

thousands of flows [8, 9]. It seemingly will be a great challenge for a scheduler to be scalable enough to serve flows well with a limited chip area.

The most popular abstraction for a programmable scheduler is the one proposed by Sivaraman et al. [7]: Push In First Out (PIFO), which ranks packets and maintains them in a priority queue. Packets can be pushed in an arbitrary position according to rank and leave from the head. By changing the rank computation function, PIFO can express a wide range of scheduling algorithms. Although the hardware design given by Sivaraman's team achieves the line rate, its scale is limited by hardware resources, far from meeting the scalability requirement of next-generation networks.

Essentially, in the hardware implementation of PIFO, what we need to perform the scheduling algorithms is a priority queue (PQ), which has been extensively studied. Existing solutions include binary trees [10], binary heaps [11–15], shift registers [10, 16, 17], systolic arrays [10, 18], register-based arrays and calendar queues [19], as well as the hybrid methods or variants. Among them, the systolic arrays like single-instruction–multiple-data (SIMD) PQ architecture [20] is the fastest, whose throughput is 10x that of PIFO's original design, but the scale is not improved a lot. pHeap [15] can support 2^{17} flows while serving only 10 Gbps traffic. Apparently, it is hard to construct a priority queue with high throughput and large scale.

In order to combine the large scale and high throughput, a large number of approximate implementations of the PIFO model have sprung up in recent years, mainly using First-In-First-Out (FIFO) queue(s) [21, 22] and Calendar Queue (CQ) [23–26] to model the behavior of PIFO. However, at any time, the ranks of packets in the scheduler will typically fall within a limited range of values [27], which is dependent on scheduling policy and traffic load. These methods require fine tuning of parameters and may face reordering under some circumstances, leading to weaker performance guarantees [8, 28].

Previous researches either approximate PIFO with some rank reorders, or cannot combine high speed and large scale. To build an accurate, large-scale and high-throughput PIFO queue, we propose the Balanced Multi-way sorting tree (BMW-Tree) and present 2 hardware designs. Any work that uses the PIFO programmable scheduler model [29–34] can benefit from our design, and we believe it is a promising component for a next-generation traffic manager.

We summarize the contributions of this paper as follows:

- A novel data structure called Balanced Multi-way sorting tree (BMW-Tree) is designed to realize the PIFO model at high speed. It is modularized, insertion-balanced and pipeline-friendly with autonomous nodes, contributing to simple and efficient hardware designs.
- We provide two hardware designs: register-based (R-BMW) and RPU-driven (RPU-BMW) BMW-Tree. Both of them are pipelined, and their frequencies are independent of the number of levels when hardware resources are affluent, owing to the modularized and autonomous nodes. R-BMW has high throughput, and a push-pop consecutive operation sequence costs R-BMW 2 cycles. To improve the scalability, we introduce Ranking Processing Units (RPU) for data processing and store tree nodes in SRAMs. RPUs run in the pipeline to maintain high throughput, and it can process a push and a consecutive pop every three cycles. With only a few RPUs, RPU-BMW can support a large number of packets.
- The proposed hardware designs are implemented in Verilog, targeting a Xilinx Alveo U200 Data Center Accelerator Card with an XCU200 FPGA. Their performance is evaluated and compared with the original PIFO implementation. Thanks to the modularized and pipelined design, R-BMW features an impressive throughput, which is 4.8 times that of the original PIFO implementation. An 11-level 2-way R-BMW is reported to reach 192 Mpps (million packets per second) for 4k flows. As for RPU-BMW, it has a remarkable capacity with the RPU-driven structure. The 8-level 4-way RPU-BMW supports 87k flows at 93 MHz. When the capacity is similar, RPU-BMW has a comparable throughput to R-BMW. Overall, RPU-BMW has both a high throughput and a large scale.
- To further validate the efficacy of RPU-BMW, we synthesize its Verilog code in GlobalFoundries 28 nm process with off-chip memory. 8-level 4-way RPU-BMW supports 87k flows at 200 Mpps, consuming a mere 0.522% (1.043mm^2) chip area and 0.57 MB off-chip memory. With an average packet size of 512 bytes, RPU-BMW reaches over 800 Gbps line rate. RPU-BMW is the first accurate PIFO implementation that supports more than 80k flows at as fast as 200 Mpps. The project is available at https://github.com/BMWTree/BMWTree.

The rest of the paper is organized as follows. We start with the background and introduce some classical scheduling algorithms and PIFO model in Section 2. Then, in Section 3, the data structure and algorithms of BMW-Tree are displayed. What follows are two pipelined hardware designs: register-based BMW-Tree in Section 4 and RPU-driven BMW-Tree in Section 5. Section 6 provides experimental verification for hardware performance. A review of prior related work is shown in Section 7. Finally, we conclude the paper in Section 8.

This work does not raise any ethical issues.

2 BACKGROUND

2.1 Scheduling Algorithms

Over the past several decades, a great many packet scheduling algorithms have been designed according to various application requirements. In general, scheduling algorithms can be classified into work-conserving and non-work-conserving. As long as the link is idle and the queue is non-empty, a work-conserving algorithm will schedule a packet to the egress port. The most prevalent among them include Weighted Fair Queueing (WFQ) [1], Deficit Round Robin (DRR) [2] and Start Time Fair Queueing (STFQ) [35] for fairness, Shortest Remaining Processing Time (SRPT) [3] for minimizing flow completion time (FCT), FIFO+ for small packet delay, to name but a few. In contrast, a non-work-conserving algorithm sets the time for packet departure, which may pend the packet and leave the link idle to shape the traffic [9, 36]. The representative scheduling algorithms include Token Bucket (TB) [4] and Stop-and-Go Queueing [37].

With the emergence of new application scenarios and the continuous upgrading of application requirements, new scheduling

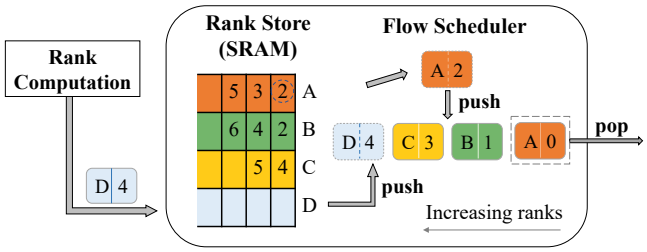


Figure 1: The diagram of PIFO Block

algorithms constantly turn up. MQFQ [5] presents a fair, work-conserving scheduler suitable for multi-queue systems. HCSFQ [6] is able to approximate hierarchical fair queueing, which naturally captures the traffic in today's data centers. Meanwhile, there are still many scenarios calling for appropriate scheduling algorithms, such as distributed computing. Take distributed machine learning as an instance, training results of different computing nodes in each round are transmitted and synchronized through the network. The long-tailed node transmission time prolongs the round and slows down model training. However, traditional packet scheduling algorithms cannot schedule distributed machine learning traffic well, and the customized scheduling algorithms are on the way.

2.2 PIFO Model and Block

Routers and switches have traditionally used specialized network ASIC chips to forward network traffic at high speeds. However, the functions of these ASICs are fixed after production and cannot be modified. Supporting new scheduling algorithms requires the design of a new ASIC, which can be time-consuming. In light of the proliferation of scheduling algorithms, it is imperative to develop a programmable scheduler. Several abstractions for scheduling have been proposed since 2016, and PIFO is the most widely discussed.

The scheduling algorithm determines the relative order or transmission time of packets when flows contend for a link. In the PIFO model, each packet is tagged with a rank by the rank function, which differentiates flows according to the QoS or QoE requirements. A smaller rank indicates a higher priority, and the packet with the smallest rank wins the contention and goes first. WFQ employs virtual departure time as rank, SRPT employs the remaining flow size as rank, and First Come First Serve (FCFS) employs arrival time as rank. These packets are maintained in a PQ.

As packets of the same flow are transmitted in FIFO order, the actual contenders are the head packets of flows, and there is no need to maintain all the packets in the queue. This means that the number of flows supported by the PQ equals the number of elements supported. As shown in Figure 1, PIFO design builds the rank store in SRAM to buffer the non-head packets of flows in order, and utilizes the flip-flops to implement the PQ (the so-called flow scheduler). Logical PIFOs and metadata are not shown for simplicity. We follow the architecture of PIFO and focus on improving the flow scheduler implementation. Other components are not optimized compared with the PIFO model [7].

2.3 Operations for Flow Scheduler

PQs provide the following operations to serve as a flow scheduler:

- push: As long as a flow is non-empty and no packet of the flow is in the PQ, insert the head packet into the PQ according to the rank.
- pop: When the link is idle, send the packet with the smallest rank to the egress port and delete the packet from the PQ.

Figure 1 gives examples of the push cases and the pop case. We use $p(x, y)$ to refer to a packet, where x is the flow ID, and y is the rank. Packet $p(A, 0)$ has the highest priority, and it belongs to flow A. After the scheduler pops it out to the egress port, the new head packet of flow A $p(A, 2)$ will be pushed into the PQ and finds its place between $p(B, 1)$ and $p(C, 3)$ according to the rank. Another case of push is that a flow D goes from empty to non-empty, then the head packet $p(D, 4)$ bypasses the rank store and enqueues the PQ.

3 THE BALANCED MULTI-WAY SORTING TREE

To obtain a large-scale and high-throughput PIFO queue, we propose a novel data structure called BMW-Tree, which is designed specifically for efficient hardware implementation. In this section, we first present the data structure and its operations as a priority queue. Then the characteristics of BMW-Tree are highlighted. Finally, we compare BMW-Tree with similar data structures, such as heaps and their variants.

3.1 Data Structure

Definition. A BMW-Tree of order M is a tree that satisfies the following properties:

- Each node of a BMW-Tree contains up to M elements.
- A non-leaf node has M pointers to M children (sub-trees). The $i-th$ sub-tree is rooted at the $i-th$ element $(1 \leq i \leq M)$ in the node. In other words, each element is the root of the sub-tree below it.
- Values of the M elements in a node are not sorted.
- Heap property is satisfied. The value of the root element is always smaller than or equal to any values below it on the sub-tree.

Element. Each element represents a packet and contains three fields as given below:

- Value: this field holds the priority of a packet.
- Counter: this field maintains the number of elements in the sub-tree rooted at the current element, including the element itself. The initial value is 0.
- Metadata: this field records the meta data of a packet.

BMW-Tree is maintained according to the values (priorities) only. ***For convenience, values and elements are used alternately to indicate a packet when there is no ambiguity.***

We use $L-M$ BMW-Tree to indicate an M-order BMW-Tree with L levels. The number of elements supported by a $L-M$ BMW-Tree is $\frac{M(1-M^L)}{1-M}$. The root node of the BMW-Tree contains two meta-information of the tree: the smallest element and the total number of elements stored. The smallest element is the element with the

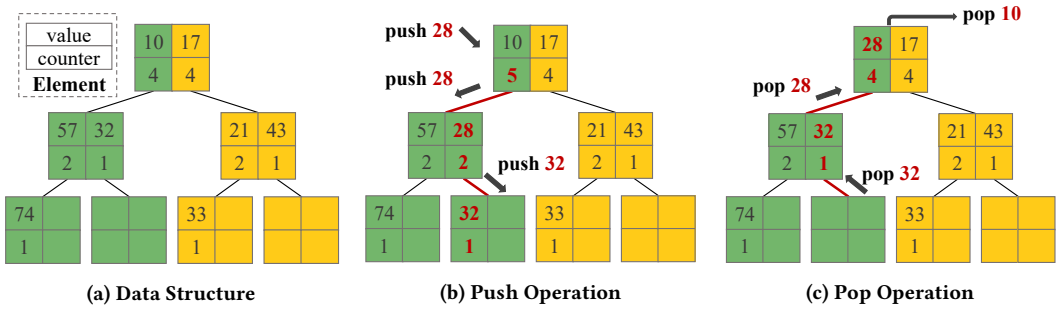


Figure 2: Example of A 3-level 2-way BMW-Tree

minimum value in a BMW-Tree. As the heap property is satisfied, It can be found in the root node. While the total number of elements can be obtained by adding counters in the root node.

Figure 2 (a) displays a 3-2 BMW-Tree, which has 7 nodes with a capacity of 14 elements. It is composed of 2 sub-trees, and they are distinguished by different colors. Metadata is omitted for brevity. A total of 8 elements are stored, with both sub-trees containing four elements.

3.2 Algorithms

The following are the algorithms for priority queue operations.

Push X: A new value X must be inserted into the BMW-Tree without breaking the properties of the tree. The procedure is as follows.

(1) If the root node is not full, place the value X in the leftmost empty space of the root node, and the counter is increased to 1. Otherwise, we select the least-loaded sub-tree to keep the tree balanced, whose counter is the smallest. If there are multiple such sub-trees, we select the leftmost to push. Suppose that the $i-th$ sub-tree is chosen, and we add the counter of its root element by 1.
(2) Compare X with the root value Y of the $i-th$ sub-tree. If X is smaller, kick Y out to the next-level node in the sub-tree, and X becomes the new root value. Otherwise, push X to the next-level node of the $i-th$ sub-tree.
(3) Repeat the push operation recursively on lower-level nodes until the pushed value finds its place.

Pop: The principle of pop is to dequeue the smallest element on a non-empty BMW-Tree.

(1) Values stored in the root node are compared and the smallest element in the root is popped out. Assuming it is the $i-th$ element $(1 \leq i \leq M)$), its counter minus 1.
(2) Elements in the next level of the $i-th$ sub-tree are compared, and the smallest one is lifted to fill the $i-th$ element of the root node.
(3) The same procedure will be applied to lower-level nodes recursively until the child is empty.

Figure 2 (a) shows the status of BMW-Tree after pushing the values of 10, 17, 57, 21, 32, 43, 74 and 33 in sequence. Supposing that there is a push 28 operation and then a pop operation on the BMW-Tree.

Because the root node is full, a value has to be pushed into the second level. The counters of the two sub-trees are (4,4), so the first sub-tree is chosen according to the push algorithm, whose root element is 10. 28 is larger than 10 and therefore pushed down to the second level of the tree. The node in the second level is also full, and the counters are (2,1), so its second sub-tree is chosen. 28 is smaller than 32, so it replaces 32, pushing 32 along the second sub-tree. Finally, 32 finds an empty position in the third level, as shown in Figure 2 (b). The corresponding counters increase by one.

Figure 2 (c) shows the pop operation. The BMW-Tree compares elements in the root node and finds that 10 is the smallest. After 10 is popped, the smallest element in the sub-tree is 28, so 28 is lifted. 32 is lifted from the third level to the second level to fill the vacant position. The corresponding counters decrease by one.

3.3 Characteristics of BMW-Tree

Here we summarize the characteristics of BMW-Tree and reveal why it is suitable for implementing large-scale, high-throughput PIFO models.

First of all, our sorting tree is modularized. All nodes are the same, and each node is only connected to parent and child nodes. Trees of various sizes can be elegantly constructed by duplicating the node and connecting them as the tree structure. This design simplifies the implementation and test.

Secondly, BMW-Tree is insertion-balanced, which indicates that the insertion of each new packet moves the BMW-Tree towards the optimum balance. Although successive pops on the same sub-tree can locally unbalance the tree, it tends to be balanced with the arrival of new packets. By inserting each new entry into the least-loaded sub-tree, all elements of BMW-Tree can be filled if we want. Thanks to the unsorted design, the insertion-balance is maintained easily.

Thirdly, the simple push and pop operations in BMW-Tree make it pipeline-friendly in hardware implementation. Data are unidirectionally transferred between adjacent levels. Push operations transfer data from top to bottom, and pop operations transfer data in the opposite direction. The time complexity of both operations is $O(log_M N/M)$ with N the number of values. But the adjacent level-only transmission enables the BMW-Tree to run in a pipeline easily.

Last but not least, nodes in BMW-Tree are autonomous. Operations on each node of the tree are independent of the rest of the

Table 1: Comparison with Heap Variants

DS \ Property	Balanced	Pipeline-friendly	Autonomous
BMW-Tree	Insertion-Balanced	✓	✓
pHeap	×	✓	×
Pipelined Heap	Self-Balanced	×	×

sub-tree. A node decides the recursion and the sub-tree to operate on its own without comparing with other nodes. The sub-tree to push is decided by the element counters, and the sub-tree to lift a value is decided by the smallest element in the node.

In a nutshell, BMW-Tree is a modularized, insertion-balanced and pipeline-friendly tree with autonomous nodes. With modularized and autonomous nodes, the frequency of the hardware implementation of BMW-Tree is not affected by the number of levels within the pipeline when hardware resources are affluent.

3.4 Comparison with Heap and Heap Variants

pHeap and Pipelined Heap are two scalable priority queues based on the traditional heap structure [14, 15]. Although BMW-Tree may resemble a D-ary Heap at first glance, it differs significantly from traditional heaps and is more suitable for large-scale and high-throughput PIFO queue implementation.

Table 1 shows the comparison of BMW-Tree with pHeap and Pipelined Heap, and DS is short for data structure. In order to support the pipeline, pHeap modifies the push and pop operations, and Pipelined Heap modifies the push operation based on the traditional min-heap. However, they still have room for improvement.

(1) pHeap is not balanced. It inserts data into the left subtree first, which will cause the left subtree to be much deeper than the right subtree.

(2) Pipelined Heap retains the pop operation of traditional min-heap, which makes it complex to pipeline. The pop operation in a heap will pop the root and place the right-most leaf at the root. Then the shift-down operation will be performed until the heap property has been re-established. During a pop, the value has to fly from bottom to top and then cross from top to bottom. The pop operation of Pipelined Heap requires each level to be connected to the root node. Besides, the youngest-in-progress insert operation has to be tracked when multiple operations are in progress in various pipeline stages. The pipeline design is expensive [14].

(3) Nodes in pHeap and Pipelined Heap need to be compared with other nodes to decide the branch. The pop operation in Pipelined Heap compares a node with its two children. The pop operation in pHeap compares a node's two children. As for push operation, pHeap has to look up the capacity of the left child first to decide whether steer insertions left or right.

BMW-Tree exhibits nice properties. However, it is a challenge to realize an accurate high-speed PIFO model with a scale of tens of thousands. In Section 4, we present a register-based hardware design. Evaluation in Section 6 shows that, when implemented atop the same FPGA, the maximum capacity of register-based BMW-Tree is about the same as the original PIFO implementation [7], but the throughput is 4.8 times higher. To further increase the scale, we propose an RPU-driven BMW-Tree in Section 5, where we store elements in SRAMs and drive the BMW-Tree with Ranking Processing Units. RPU-driven BMW-Tree has both large scale and high throughput.

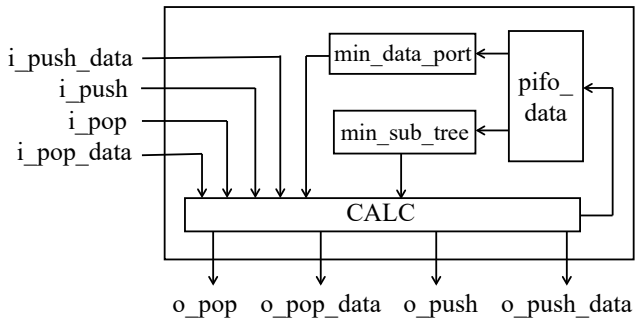


Figure 3: The Sketch of R-BMW Building Block

4 REGISTER-BASED BMW-TREE

To accelerate the scheduler and reduce the complexity of hardware design, a register-based BMW-Tree (R-BMW) is proposed. All nodes are implemented by registers, and the R-BMW is modularized and pipelined. In this section, we introduce the R-BMW hardware design. We first abstract nodes as modular building blocks. Then the pipelined BMW-Tree is shown, specifying the node behavior in each cycle. Finally, the summary of R-BMW is given.

4.1 Modular Building Block

As mentioned in Section. 3.3, BMW-Tree is modularized. To build the R-BMW, we only have to design the modular building blocks and connect them. A node only connects the nodes of the adjacent level(s) instead of connecting each level to the first level as Pipelined Heap [14] does. Modularization also simplifies upgrades, which allows us to optimize the entire tree by optimizing the building block.

We analyze the interfaces required for push and pop in the BMW-Tree structure, and find that the atomic operations of each node are identical. The element entering the node comes from the packet pushed from the higher level, or is lifted from the lower level node. The element leaving the node is pushed to the lower level or popped to the higher level. The building block is designed as Figure 3.

The pins of the building block are listed below. Pins of *clk*, *push_available*,*pop_available*, and *almost_full* are not drawn in Figure 3 for brevity. Unless otherwise stated, the width is 1 bit. The 'i' prefix indicates input and 'o' prefix indicates output.

- *i_push/i_pop*: the enable signal of push/pop received from the parent or the external.
- *i_push_data/i_pop_data*: the element to insert which is from the higher/lower level. The width is the length of priority and metadata.
- *clk*: the clock signal.
- *o_push/o_pop*: the enable signal to the child for push/lifting an element. The width is M bits.
- *o_push_data*: the element sent to the child. The width is the length of priority and metadata.
- *o_pop_data*: the smallest element sent to the parent or the external. The width is the length of priority and metadata.

- *almost_full*: the signal indicates whether the entire tree is about to be full. When the signal is set to 1, no new push operation should be issued.
- *push_available/pop_available*: the signal is used to indicate whether the current cycle can handle a new push/pop operation correctly.

A node consists of four components: *pifo_data*, *min_sub_tree*, *min_data_port* and *CALC*. Elements are stored in *pifo_data*. During a push operation, *min_sub_tree* calculates the port of the least-loaded sub-tree and sends its root element to *CALC*, where the root element is compared with *i_push_data*. The larger one will be pushed to the next level. During the pop operation, *min_data_port* determines the port of the smallest element, and transmits it to *CALC* for output.

The *CALC* module is also responsible for the determination of *almost_full*, which is easy because the total capacity of BMW-Tree and the number of stored elements are readily available as described in Section 3.1. The assignment of available signals is also accomplished by the *CALC* module, and we will introduce it in Section 4.2.2.

4.2 Pipelined R-BMW

Although the time complexity of BMW-Tree is $O(log_M N/M)$, with the pipeline, we can achieve O(1) amortized time operations in hardware. Firstly we introduce the pipelined R-BMW, which drives signal and data transmission in sequential logic to show the steps of push and pop operations. Then we optimize its throughput by sustained transfer.

4.2.1 Operations

The R-BMW is ready to handle a new operation when a node finishes the precedent operation. Nodes in the different levels perform operations concurrently.

A node needs three cycles to finish a pop operation. In the first cycle, the node pops its smallest element. Supposing the $i-th$ element is the smallest, a pop signal is sent to the $i-th$ child. In the second cycle, the $i-th$ child pops its smallest element to the parent node. In the third cycle, the parent gets the element lifted from the child. After a node finishes a pop operation, it can perform another pop/push operation with correctness guaranteed. With the pipeline, pop operations can be issued every three cycles.

For the push operation, the node in each level decides which element should be recorded in the current node and passes the other to the next level in a cycle. Then it finishes the push and is ready for any new operation with correctness guaranteed. With the pipeline, the issue rate of push operations is one per cycle.

Although the R-BMW is simple and elegant, its throughput needs improving due to the three clock cycles required for each pop operation. After popping the smallest element, the substitution lifted from the child node arrives too late. If a new pop operation is issued before the fourth cycle, R-BMW fails to handle it correctly because of the wrong state of the node.

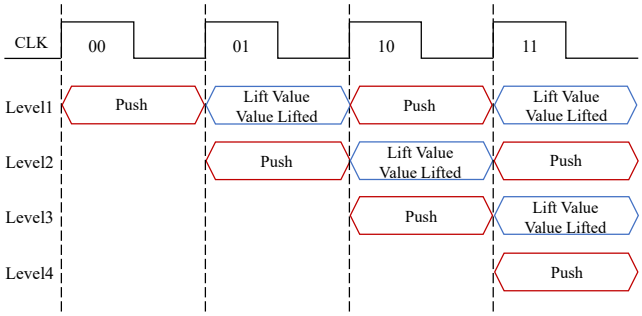


Figure 4: Timing Graph of R-BMW with Push-Pop Sequence

4.2.2 Sustained Transfer

In order to improve the throughput of R-BMW, we employ sustained transfer and turn the data transmission into combinatorial logic. All nodes consistently report their smallest element to the parent regardless of whether they receive the pop signal. In this way, within the clock when a node receives the pop signal, it can obtain the children's smallest elements. The parent node calculates its own smallest element, pops it, and selects the corresponding child's smallest element as a substitute. With correct data prepared, the element in the parent node is accurate on the next rising edge of the clock.

It is worth noting that although the node's element is correct in the second cycle after the pop signal reception, the child node is still performing the pop operation. To ensure obtaining the right elements from the next level, a new pop operation can only be issued in the third cycle. As the completion of a push operation does not depend on child nodes, a new push operation can be issued following a pop.

In summary, R-BMW is capable of handling push-push and push-pop (pop-push) sequences, but not two consecutive pop operations. To maintain the correctness, the *pop_available* signal is set to 0 after receiving a pop signal, which indicates a pop immediately after a pop is illegal. It turns into 1 after a push or null signal. *Push_available* always keeps 1, as the node can correctly process a new push operation every cycle. Figure 4 depicts the timing graph of the optimized R-BMW with the push-pop sequence, and a node can deal with a consecutive push and a pop operation every two cycles with the pipeline.

For the implementation of sustained transfer, We make two modifications to the building block shown in Figure 3 for R-BMW. (1) A new pin *o_pop_result* is added as the external interface for the BMW-Tree output. Its width is the length of priority plus metadata. The sustained transfer leads to nodes continuously popping data to a higher level. To avoid unnecessary pops of the root node through *o_pop_data*, we employ *o_pop_result* to output for the root node. When *i_pop* in the root node is set to 1, the element of *o_pop_result* is equal to the element of *o_pop_data* and popped out of BMW-Tree. *o_pop_result* only works in the root node of BMW-Tree, and *o_pop_data* is used in the non-root node to transmit data to the parent node in real time. (2) The width of *i_pop_data* is changed into M times the length of *o_pop_data*.

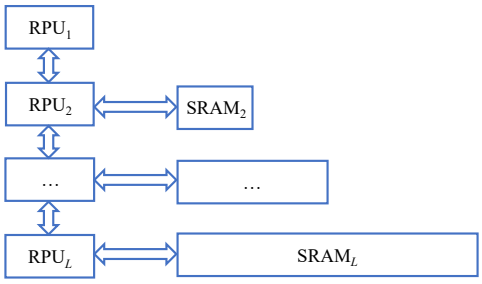


Figure 5: RPU-driven BMW-Tree

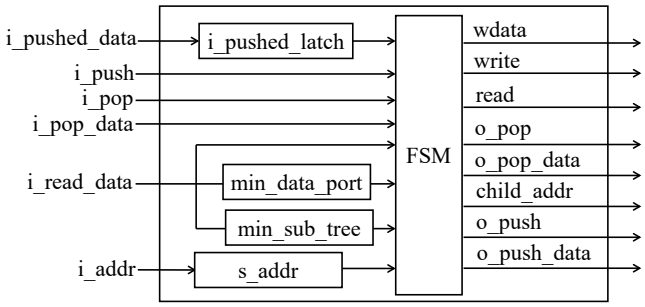


Figure 6: The Sketch of RPU in RPU-BMW

4.3 Takeaway

The R-BMW is totally modularized, and each node connects the parent and children only, greatly reducing the hardware complexity. With the pipeline, the R-BMW can correctly issue one push per cycle and one pop every two cycles. All sequences of consecutive operations are supported except for consecutive pop operations. The most common push-pop successive sequence takes a node 2 cycles to finish a push and a pop. R-BMW has a high clock frequency independent of the number of levels with the pipeline in theory. The performance of R-BMW will be evaluated in Section 6.

5 RPU-DRIVEN BMW-TREE

The register-based implementation of a BMW-Tree may take lots of registers which greatly limits the capacity for elements. We introduce the RPU-driven BMW-Tree (RPU-BMW) to save register resources and improve scalability. Firstly, we will introduce the Ranking Processing Unit (RPU) and give a description of the RPU-driven architecture. Then a plain version is presented, revealing the steps of push and pop operations. Afterwards we propose two tricks, which enable RPU-BMW to handle a push operation in each clock cycle and a pop operation every two clock cycles. Finally, we summarize the RPU-BMW design.

5.1 Architecture of RPU-driven BMW-Tree

Since BMW-Tree is modular and each node is autonomous, we design a Ranking Processing Unit to complete the computing tasks of each node. To reduce register usage, we store nodes of each level in an SRAM, and perform operations using an RPU. Nodes in memory will be dynamically loaded into/off from RPU and time-share the RPU, similar to how programs share a CPU.

During the pipeline process, there is only one active node in each level at any time. In a bid to boost the system throughput, we can set one RPU for each level of the BMW-Tree, and each RPU will be shared by nodes on the same level. The modular architecture is depicted in Figure 5. RPU_i is shared by nodes in the $i-th$ SRAM. As the root node is the sole node in the first level, it exclusively occupies the first RPU. In the lower levels, only the nodes accessed by the current push or pop operations are placed in the corresponding-level RPUs. When RPU_i completes the operation, the node will be written back to $SRAM_i$. As long as the values in $SRAM_i$ are correct, RPU_i can run a new operation. Data from different levels interact through the RPU array. Multiple RPUs run in a pipeline. Compared with register-based implementation schemes, the RPU-driven BMW-Tree combines L RPUs with $L-1$ SRAMs. Considering that the capacity of BMW-Tree grows exponentially with the number of levels L, we only need a few RPUs to realize a large-scale BMW-Tree.

The modular ranking processing unit is shown in Figure 6. It is similar to the building block in Figure 3 but has the following structural features.

(1) The ranking processing unit only connects to its predecessor and successor. Namely, RPU_i just connects to RPU_{i-1} and RPU_{i+1} except the first and last level.

(2) The *i_pushed_latch* stores the pushed value temporarily and waits for the data to be caught from SRAM.

(3) In order to interact with SRAM, 6 wires are added to the RPU: write enable signal *o_write*, read enable signal *o_read*, write data *o_write_data*, read data *i_read_data*, write address *o_child_addr* and read address *i_addr*.

(4) Because the nodes of the same level share a piece of SRAM, it is necessary to add the control logic *s_addr* for indexing the node address in the SRAM. Suppose the node in RPU_i is the $j-th$ node in $SRAM_i$ and it has to emit signals to RPU_{i+1} for its $k-th$ child node, the address in $SRAM_{i+1}$ is $(j-1)\times m+k$.

(5) *FSM* serves a similar function as the *CALC* module in a R-BMW node, and pins of *clk*, *push_available/pop_available*, and *almost_full* are omitted as well. FSM determines *almost_full* in the same way as a R-BMW node. The assignment of *push_available/pop_available* is introduced in Section 5.2.3.

5.2 Pipelined RPU-BMW

5.2.1 Operations

The design in sequential logic is introduced to reveal the details of push and pop operations.

An RPU costs 3 cycles for a push operation. In cycle 1, the push signal arrives at the $i-th$ level. The read signal is sent to the $i-th$ SRAM for the $j-th$ node which is the least-loaded, and j is assigned by the $(i-1)-th$ level ($i>1$). The input value is retained in the RPU_i. On the rising edge of cycle 2, RPU_i reads the values from the SRAM, and compares the least loaded one with the reserved push value. When the new value for the $j-th$ node is determined, the push signal and the push value are sent to the next level. In the third cycle, send the write signal and the $j-th$ node to the $i-th$ SRAM.

For the pop operation, an RPU costs 6 cycles. In cycle 1, the pop signal arrives at the $i-th$ level. A read signal is sent to the $i-th$ SRAM for the $j-th$ node. j is assigned by the $(i-1)-th$ level ($i > 1$). On the rising edge of cycle 2, the values of the $j-th$ node are read into RPU_i, and the minimum value is calculated. In cycle 3, supposing the popped value is from the $k-th$ element, the pop signal is sent to the $(i+1)-th$ level for the $k-th$ node. The RPU_{i+1} sends read signal to the $(i+1)-th$ SRAM. On the rising edge of cycle 4, the values of the $k-th$ node are read into RPU_{i+1}, and the minimum value is calculated. In cycle 5, the minimum value is lifted from RPU_{i+1} to RPU_i. On the rising edge of cycle 6, RPU_i receives the value and fills the $k-th$ element of the $j-th$ node. RPU_i sends the write signal to the $i-th$ SRAM.

Improving the data processing capability of RPU is a challenge. To reduce the number of operation cycles, we propose two tricks: change the sequential logic into combinatorial logic and hide operations using Simple Dual Port RAMs.

5.2.2 Combinatorial Logic

We turn all the logic of RPU into combinatorial logic. Thus, the number of cycles is squeezed. For the push operation, on the rising edge of the first cycle, RPU_i still receives the push signal and tries to read SRAM. On the rising edge of cycle 2, RPU_i gets the node from the SRAM, compares the least loaded element with the new element, pushes the larger one to the next level, and writes the correct node back to RPU_i. Push operations can be issued every 2 cycles.

For the pop operation, the first cycle is the same as before. On the rising edge of cycle 2, RPU_i gets the node from SRAM, calculates the minimum value, pops the value, and sends a pop signal to the next level. On the rising edge of cycle 3, RPU_i receives the lifted value from the $(i+1)-th$ level and fills the value in its node. The node is written back to the SRAM. Pop operations can be issued every 3 cycles.

5.2.3 Operation Hidden

The SRAM used in our project is Simple Dual Port RAM with separate read/write addresses and a single read/write clock. It has such a property that if read and write operations are performed on the same address simultaneously, the newly written data will be returned. For instance, the original value of an element is 32 and the new value is 28. Supposing writing 28 into SRAM (a finished push) and reading the value from SRAM (a new push) occur in a cycle. 28 will not be latched until the next rising of the clock, but the read in the current cycle will get 28, which is correct for the new push operation.

We accelerate the push and pop operations one step further by exploiting the property. The property provides correctness for issuing a read when the write is in process. Since we write to the SRAM in the last cycle of each operation and read from the SRAM in the first cycle of each operation, we can issue an operation one cycle ahead. Namely, it is correct that we overlap any two consecutive operations. Take two push operations as an example. If the read and write share the same address, the second push issued at cycle 2 will read the right value which the first push should write. As a result, we can issue one push operation per cycle. For the same reason, a new pop can be issued at the third cycle of the old pop, which means our architecture supports one pop every two cycles.

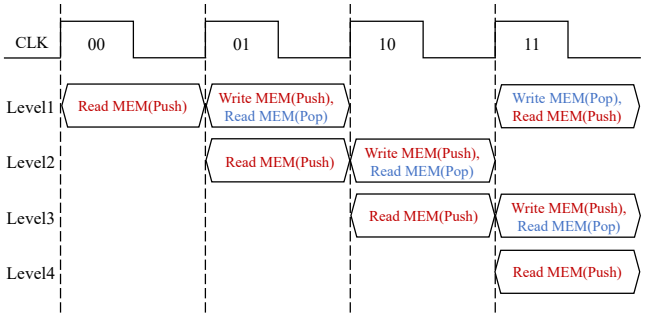


Figure 7: Timing Graph of RPU-BMW with Push-Pop Sequence

Thanks to the Combinatorial Logic and Operation Hidden, a push operation followed immediately by a pop can also be processed correctly. However, pop-push and pop-pop sequences are not supported. There must be an idle cycle after a pop. Upon receiving a pop signal, the RPU sets *push_available* and *pop_available* to 0, and turns them into 1 after receiving a push or null signal. Figure 7 depicts the timing graph of RPU-BMW with push-pop consecutive sequences and shows that three cycles are needed.

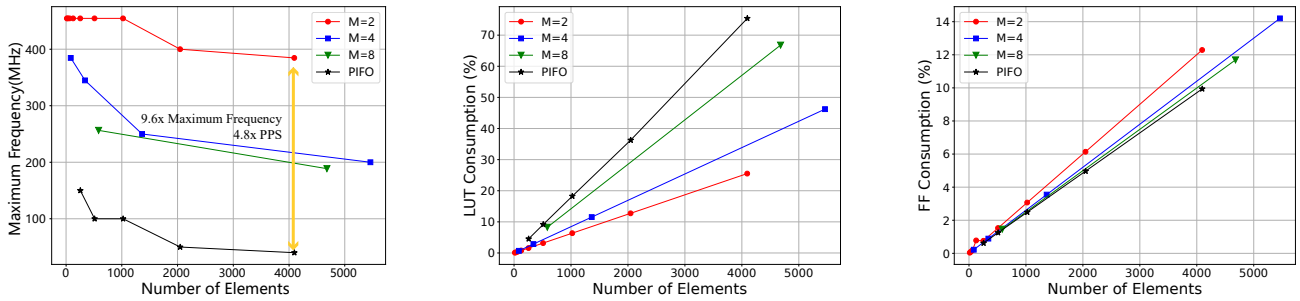
5.3 Takeaway

The RPU-BMW consists of modular RPUs and SRAMs. Compared with R-BMW, RPU-BMW saves a great number of registers and obtains large scale by introducing ranking processing units and putting nodes in SRAM. With the pipeline, RPU-BMW can handle a push per cycle and a pop every two cycles. An idle cycle is required after a pop. The most common push-pop successive sequence takes an RPU 3 cycles to finish a push and a pop. RPU-BMW has a high clock frequency independent of the number of levels with the pipeline in theory. The performance of RPU-BMW will be evaluated in Section 6.

6 IMPLEMENTATION AND EVALUATION

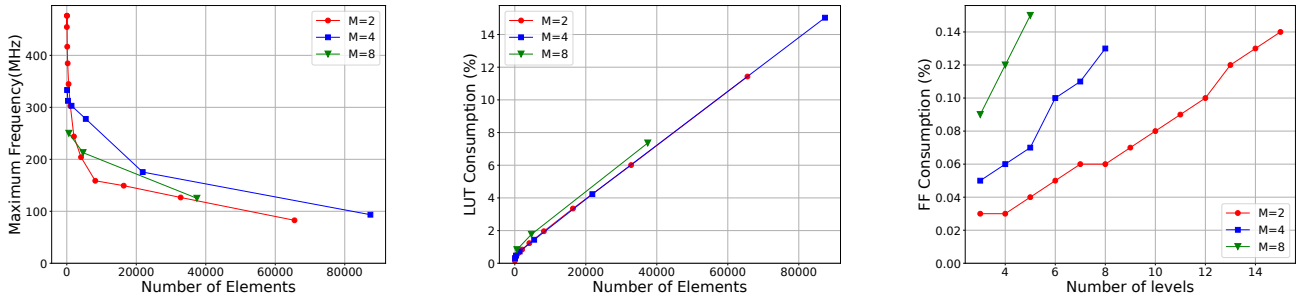
The prototype of Register-based pipelined BMW-Tree and RPU-driven pipelined BMW-Tree is written in Verilog and implemented on Xilinx Alveo U200 Data Center Accelerator Card with XCU200 FPGA, which has 1182k LUTs, 591k LUTRAMs and 2364K flip flops. **We also synthesized the open-source PIFO implementation [7] to serve as a baseline, referred to as PIFO for convenience.** The value (priority) is set as 16 bits and the metadata 32 bits, which are the same as in [7]. We test how many flows can be supported in these schemes. The clock frequency and resource consumption are obtained through the Vivado synthesis tool.

Adjusting the order (M) and the number of levels (L), a BMW-Tree can be created as long as enough resources are available. The number of elements supported by a BMW-Tree is $\frac{M(1-M^L)}{1-M}$. R-BMW is composed of $\frac{(1-M^L)}{1-M}$ building blocks. RPU-BMW is composed of L ranking processing units and $L-1$ SRAMs. For $M = 2, 4, 8$, we



(a) Maximum Frequency (b) Percentage of LUT Consumption (c) Percentage of FF Consumption

Figure 8: Comparison of R-BMW and PIFO on FPGA



(a) Maximum Frequency (b) Percentage of LUT Consumption (c) Percentage of FF Consumption

Figure 9: Comparison of RPU-BMW among Different M on FPGA

measure the frequency and resource consumption from 3 levels to the highest level supported by our FPGA.

We also synthesize the Verilog code of RPU-BMW in GlobalFoundries 28 nm process, and obtain the performance and area cost through Design Compiler. Components such as pointers are not included.

Finally, the packet level evaluation is given.

6.1 Cost and Performance of R-BMW

As mentioned in Section 2.2, the number of flows supported by the priority queue equals the number of elements supported. Both the LUT and FF resource consumption of R-BMW and PIFO are linearly related to the number of elements. Limited by the LUT resources, a 2-order R-BMW can be scaled up to 11 levels with a capacity of 4094 flows.[1] The 11-2 R-BMW can run as fast as 384 MHz, which reaches 192 Mpps. PIFO shares a similar scalability with the 2-order R-BMW, which supports 4096 flows. However, its maximum frequency is only 40 MHz and it can only process packets at 40 Mpps. The throughput of R-BMW is 4.8 times that of PIFO. The maximum capacity of 4-order R-BMW is 5460 flows, with a frequency of 200 Mhz. The 8-order R-BMW supports 4680 flows at 188 Mhz. Their throughput is 2.5 times and 2.35 times that of PIFO, respectively.

[1]Theoretically, resources on the FPGA board are enough to support a 12-2 R-BMW. However, it takes too long to implement it due to routing congestion.

R-BMW beating PIFO in throughput can be explained from two aspects. For one thing, the pushed value must be input to all of the PIFO blocks, which causes a bus loading problem and adds to the hardware cost. For another thing, PIFO finishes an operation in one cycle, while R-BMW breaks down operations into multiple cycles, greatly reducing the critical path length. With the pipeline, R-BMW can still maintain a high throughput.

Figure 8 presents the maximum frequency and resource consumption of R-BMW and PIFO with different capacities. We have the following observations.

In Figure 8 (a), the maximum frequency of a 2-order R-BMW keeps the same when the number of elements is smaller than 1022 ($L \leq 9$). As mentioned before, when the resources on the FPGA board are abundant, the maximum clock frequency is independent of the number of levels. Instead, it is related to the complexity of the node. Therefore, with a similar number of elements, the smaller M is, the simpler the node, and the larger the maximum clock frequency.

Figure 8 (b) and (c) show that LUTs and FFs cost per element are constant. The more complex the building block is, the more LUTs are consumed per element. PIFO consumes the most LUTs. The FF resource consumption per element of 2-order R-BMW is slightly larger than that of $M = 4$ and $M = 8$. The root cause lies in that, each node will consume a part of FF for caching data in addition to the elements. The additional FF consumption of each node is averaged over M elements, and 2-order R-BMW suffers a

Table 2: Performance and Resources of RPU-BMW on FPGA

M	L	Cap	Fmax	LUT(%)	LUTRAM(%)	FF(%)
2	15	65534	82.64	11.43	20.13	0.14
4	8	87380	93.45	15.03	26.81	0.13
8	5	37448	125	7.36	11.52	0.15

disadvantage with a small M. Compared with PIFO, R-BMW needs to store counters additionally, so the FF resource consumption per element is relatively more.

6.2 Cost and Performance of RPU-BMW

Table 2 lists the largest scale of RPU-BMW on the FPGA board. It has such a remarkable scale that an 8-4 RPU-BMW can support 87k elements with 26.81% LUTRAM consumed at a maximum frequency of 93.45 MHz. A 15-2 and 5-8 RPU-BMW support 65k and 37k flows, respectively. Considering that the most common push-pop successive sequence takes an RPU 3 cycles to finish a push and a pop, all of these configurations reach 100 Gbps with an average packet size of 512 bytes.

Figure 9 shows the performance and cost comparisons among different orders of RPU-BMW on FPGA. As the placement and routing get harder with the increasing number of elements, the maximum frequency decreases linearly with the number of levels. In order to show the relationship between RPU-BMW's capacity and frequency more intuitively, we set the x-axis of Figure 9 (a) as the number of elements. LUTs are consumed by both the RPUs and SRAMs. The consumption is proportional to the number of elements, regardless of the order and level, as shown in Figure 9 (b). This is because LUTs used for SRAMs are related to elements, and they account for the majority. The consumption of LUTRAM is caused by SRAMs and has a similar trend to LUTs. Due to space limits, we do not present it. Figure 9 (c) gives the percentage of FF consumed. FF is mainly consumed by ranking processing units, so its consumption grows approximately linearly with the number of levels.

We do not compare RPU-BMW with PIFO in Figure 9, as the resource consumption of PIFO is much larger than that of RPU-BMW. With a similar capacity, the frequency of RPU-BMW can be 2.5x-5x that of PIFO.

Table 3 presents a comparison of R-BMW and RPU-BMW on FPGA with the same capacities, which correspond to the largest scale for R-BMW. RPU-BMW costs much fewer resources than R-BMW. The maximum frequencies are also larger when $M = 4$ and $M = 8$ due to the affluent resources.

For R-BMW, we recommend deploying a 2-order BMW-Tree, which can achieve high throughput and is similar in scale to 4-order and 8-order BMW-Tree. While for RPU-BMW, We recommend choosing the 4-order implementation for its highest throughput and excellent scalability. The number of levels should depend on the resource budget and the application demand.

6.3 28nm-ASIC Implementation

The Verilog code of RPU-BMW is also synthesized in the GlobalFoundries 28 nm process (GF28) process. In production, external SRAMs can be used to expand the capacity, and we employ them to store nodes for simulation. To avoid increasing the complexity of chip pins, external SRAMs are only used for $SRAM_{L-1}$ and $SRAM_L$, while $SRAM_2 \sim SRAM_{L-2}$ are placed on the chip. Scattered LUT resources can be used to form several SRAMs of different sizes, and the different sizes in each level will not cause problems. SRAMs can be as fast as 800 MHz, which means they will not become the frequency bottleneck. We also synthesize PIFO in the GF28 process. The performance and area consumption of RPU-BMW and PIFO are shown in Table 4. The chip area measurement does not include other components except the flow scheduler. The total area of the chip is 200 mm^2, which is the same setting as in PIFO.

Since the 600 MHz RPU-BMW can reach a scheduling rate of 200 Mpps, which corresponds to over 800 Gbps with an average packet size of 512 bytes, we do not test higher frequencies as the throughput is high enough. It is worth noting that the maximum frequency of RPU-BMW in FPGA is significantly higher than that of PIFO with a similar capacity. Hence it must have a larger frequency than PIFO in the same ASIC technology.

8-4 RPU-BMW supports 87k flows with 1.043 mm^2 chip area and 0.57 MB off-chip memory. Power consumption is reported to be 5.79 mW, which is low since there is only one active path per operation and the active paths are short. 5-8 RPU-BMW supporting 37k flows has an even smaller area than a 1k PIFO (0.127 mm^2 VS. 0.404 mm^2), with power consumption 3.10 mW. To our best knowledge, RPU-BMW is the first accurate PIFO implementation that supports more than 80k flows at as fast as 200 Mpps.

6.4 Packet Level Evaluation

We implement RPU-BMW and PIFO in the packet-based simulator, NS3 [38], to evaluate BMW-Tree's performance improvement over PIFO due to the large scale. A star topology is built, with 128 source hosts sending TCP traffic to a single destination host. All the links have a bandwidth of 10 Gbps and a propagation delay of 3 ms. To evaluate the packet schedulers on the bottleneck link, we implement the RPU-BMW and the PIFO on the output link of the switch node.

To emulate real-world traffic, we generate TCP flows according to an empirical flow-size distribution collected in a data center that supports web-search service [39]. Each of the flows initiates on a random source host, and the start time of each TCP flow follows the Poisson distribution. For the congestion control of TCP flows, all the source hosts apply TCP New Reno.

We apply STFQ [35] on RPU-BMW and PIFO to provide fair bandwidth allocation on the bottleneck link among all the TCP flows. For simplicity, all the TCP flows share the same weight in STFQ. The capacity of RPU-BMW and PIFO is set to 4094 and 512, respectively.

PIFO has a 0.5% $\sim$ 4% packet loss rate in flow size intervals, leading to large average normalized FCT. As Figure 10 shows, RPU-BMW reduces the normalized FCT by 6%$\sim$20% for medium and large flows. However, for small flows, the average normalized FCT increases slightly. This is because small flows usually complete the transmission within several RTTs, but the queuing delay becomes longer due to the larger buffer.

Table 3: Comparison of R-BMW and RPU-BMW on FPGA

Parameter			R-BMW				RPU-BMW			
M	L	Capacity	Fmax	LUT(%)	LUTRAM%	FF(%)	Fmax	LUT(%)	LUTRAM%	FF(%)
2	11	4094	384.61	25.51	/	12.29	204.08	1.23	1.31	0.09
4	6	5460	200	46.222	/	14.2	277.77	1.44	1.73	0.1
8		4680	188.67	66.79	/	11.69	212.76	1.77	1.49	0.12

Table 4: Comparison of RPU-BMW and PIFO on 28 nm ASIC

M	L	Capacity	Meets Timing at 600 MHz	Chip Area / mm^2 (%)	Off-chip Mem (MB)
4	8	87380	Yes	1.043 (0.522%)	0.57
8	5	37448	Yes	0.127 (0.064%)	0.25
PIFO		1024	Yes	0.404 (0.202%)	-

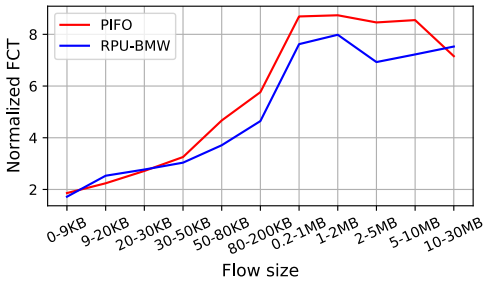


Figure 10: Average Normalized FCT

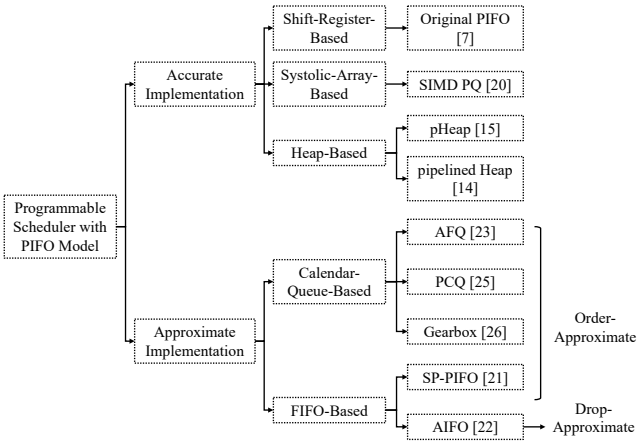


Figure 11: Classification of Programmable Schedulers

7 RELATED WORK

7.1 Packet Scheduling Model

Apart from PIFO, several abstractions are also proposed. UPS [40] claims that by appropriately initializing slacks, many different scheduling objectives can be emulated using Least Slack Time First. A packet with smaller slack should be delivered earlier. In essence, it is also a priority queue. In Eiffel [27], a generalization of the PIFO primitive in software schedulers is proposed, which can rank packets on both enqueue and dequeue. Plus, a flow also has the rank attribute and can be scheduled by the flow rank, which is tailored for algorithms like pFabric. Vishal Shrivastav [8] proposes Push-In-Extract-Out (PIEO) primitive and gives a hardware design. It allows dequeue anywhere, and the abstraction is extended to the smallest eligible packet first. We continue to focus on the PIFO and the priority queue solutions.

7.2 Hardware Solutions

Typical recent PIFO-based programmable scheduler designs are classified as shown in Figure 11. Some works [14, 15, 20] are not designed to be programmable schedulers, but the priority queues they provide can be used to implement PIFO queues.

The accurate implementation of the programmable scheduler means that packets are dequeued according to increasing rank, without out-of-order. Sivaraman et al. propose the original PIFO design [7] based on the shift register, whose scale is greatly limited by the bus loading problem and linearly growing comparator's complexity. SIMD PQ [20] has been devised, which shares a similar logic with systolic-array-based PQ. It can support links of 100 Gbps with 64-byte-sized packets and 3k flows, which is still not scalable enough for a programmable scheduler. Pipelined Heap [14] employs the conventional binary heap, and pHeap [15] devises a new tree satisfying the heap property. Both of them utilize SRAM to store nodes and have a large scale. However, pHeap suffers from wasted idle cycles and Pipelined Heap features expensive hardware designs.

It is hard to implement an accurate, large-scale and high-speed programmable scheduler. Some works propose approximate implementations that allow the dequeued packets to be inconsistent with those in a priority queue. Recently some proposals [23, 25, 26] use the calendar queue to approximate a PQ, which dequeue elements of the first non-empty bucket. AFQ [23] and Gearbox [26] only support fair queueing algorithms. PCQ [25] uses a recirculation scheme to conquer the larger priority ranges, which leads to throughput reduction on high-speed switches. FIFO queues are also utilized to approximate a PIFO. SP-PIFO [21] employs multiple FIFOs and AIFO [22] employs one FIFO. They set a bound for a queue and control the admission of packets, introducing the extra computation cost. All of AFQ, PCQ, Gearbox and SP-PIFO approximate a PIFO queue in dequeue order. AIFO approximates a PIFO queue in dropped packets.

8 CONCLUSION

In this paper, we propose a new data structure called BMW-Tree for accurate, large-scale and high-throughput PIFO implementation.

The BMW-Tree is modularized, insertion-balanced and pipeline-friendly with autonomous nodes. Based on the tree, we build two pipelined hardware designs named R-BMW and RPU-BMW. R-BMW achieves high throughput while maintaining a relatively small scale, whereas RPU-BMW features both large scale and high throughput. BMW-Tree is likely to be an attractive option for the programmable scheduler in the next-generation traffic managers.

ACKNOWLEDGMENTS

This work is sponsored by the Key-Area Research and Development Program of Guangdong Province (2021B0101400001), National Natural Science Foundation of China (62150610497, 62172108, 62002066), Natural Science Foundation of Shanghai (23ZR1404900), the Major Key Project of PCL, and Open Research Projects of Zhejiang Lab (2022QA0AB07).

REFERENCES

[1] Alan Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review* (1989).
[2] Madhavapeddi Shreedhar and George Varghese. 1996. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking (ToN)* (1996).
[3] Linus E Schrage and Louis W Miller. 1966. The queue M/G/1 with the shortest remaining processing time discipline. *Operations Research* (1966).
[4] Wikipedia. 2022. Token bucket. https://en.wikipedia.org/wiki/Token_bucket. (2022).
[5] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. 2019. Multi-Queue Fair Queuing. In *USENIX Annual Technical Conference (ATC)*.
[6] Zhuolong Yu, Jingfeng Wu, Vladimir Braverman, Ion Stoica, and Xin Jin. 2021. Twenty Years After: Hierarchical Core-Stateless Fair Queueing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[7] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable packet scheduling at line rate. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.
[8] Vishal Shrivastav. 2019. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*.
[9] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*.
[10] Sung-Whan Moon, Jennifer Rexford, and Kang G Shin. 2000. Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Transactions on computers* (2000).
[11] Muhuan Huang, Kevin Lim, and Jason Cong. 2014. A scalable, high-performance customized priority queue. In *24th International Conference on Field Programmable Logic and Applications (FPL)*.
[12] Chetan Kumar Ng, Sudhanshu Vyas, Jonathan A Shidal, RonK Cytron, ChristopherD Gill, Joseph Zambreno, and Phillip H Jones. 2012. Improving system predictability and performance via hardware accelerated data structures. *Procedia Computer Science* (2012).
[13] NG Chetan Kumar, Sudhanshu Vyas, Ron K Cytron, Christopher D Gill, Joseph Zambreno, and Phillip H Jones. 2014. Hardware-software architecture for priority queue management in real-time and embedded systems. *International Journal of Embedded Systems* (2014).
[14] Aggelos Ioannou and Manolis GH Katevenis. 2007. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Transactions on Networking (ToN)* (2007).
[15] Ranjita Bhagwan and Bill Lin. 2000. Fast and scalable priority queue architecture for high-speed network switches. In *IEEE International Conference on Computer Communications (INFOCOM)*.
[16] Ravikesh Chandra and Oliver Sinnen. 2010. Improving application performance with hardware data structures. In *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*.
[17] Gedare Bloom, Gabriel Parmer, Bhagirath Narahari, and Rahul Simha. 2012. Shared hardware data structures for hard real-time systems. In *Proceedings of the 10th ACM international conference on Embedded software*.
[18] Pierre Lavoie, David Haccoun, and Yvon Savaria. 1994. A systolic architecture for fast stack sequential decoders. *IEEE Transactions on Communications* (1994).
[19] Randy Brown. 1988. Calendar queues: a fast 0 (1) priority queue implementation for the simulation event set problem. *Commun. ACM* (1988).
[20] Imad Benacer, François-Raymond Boyer, and Yvon Savaria. 2018. A Fast, Single-Instruction–Multiple-Data, Scalable Priority Queue. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2018).
[21] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. 2020. SP-PIFO: approximating push-in first-out behaviors using strict-priority queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[22] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. 2021. Programmable packet scheduling with a single queue. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.
[23] Naveen Kr Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[24] Jingling Liu, Jiawei Huang, Ning Jiang, Weihe Li, and Jianxin Wang. 2020. Achieving High Utilization for Approximate Fair Queueing in Data Center. In *IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*.
[25] Naveen Kr Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[26] Peixuan Gao, Anthony Dalleggio, Yang Xu, and H. Jonathan Chao. 2022. Gearbox: A Hierarchical Packet Scheduler for Approximate Weighted Fair Queuing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[27] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. 2019. Eiffel: Efficient and Flexible Software Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[28] Jun Xu and Richard J Lipton. 2002. On fundamental tradeoffs between delay bounds and computational complexity in packet scheduling algorithms. *ACM SIGCOMM Computer Communication Review* (2002).
[29] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, et al. 2019. PicNIC: predictable virtualized NIC. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*. 351–366.
[30] Brent Stephens, Aditya Akella, and Michael M Swift. 2018. Your programmable NIC should be a programmable switch. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets)*. 36–42.
[31] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. 2022. Taurus: a data plane architecture for per-packet ML. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1099–1114.
[32] Ting Qu, Deke Guo, Jie Wu, Xiaolei Zhou, Xin Lu, and Zhong Liu. 2019. Efficient event scheduling of network update. *IEEE Transactions on Network and Service Management (TNSM)* 17, 1 (2019), 416–429.
[33] Wei Quan, Wenwen Fu, Jinli Yan, and Zhigang Sun. 2020. OpenTSN: an open-source project for time-sensitive networking system development. *CCF Transactions on Networking* 3, 1 (2020), 51–65.
[34] Christoph Gärtner, Amr Rizk, Boris Koldehofe, Rhaban Hark, René Guillaume, Ralf Kundel, and Ralf Steinmetz. 2021. POSTER: Leveraging PIFO Queues for Scheduling in Time-Sensitive Networks. In *2021 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. IEEE, 1–2.
[35] Pawan Goyal, Harrick M Vin, and Haichen Cheng. 1997. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on networking* 5, 5 (1997), 690–704.
[36] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. 2014. SENIC: Scalable NIC for end-host rate limiting. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[37] S Jamaloddin Golestani. 1990. A stop-and-go queueing framework for congestion management. In *Proceedings of the ACM symposium on Communications architectures & protocols*.
[38] George F Riley and Thomas R Henderson. 2010. The ns-3 network simulator. In *Modeling and tools for network simulation*. Springer, 15–34.
[39] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review* (2013).
[40] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Universal packet scheduling. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 501–521.