Formal Abstractions for Packet Scheduling

ANSHUMAN MOHAN, Cornell University, USA
YUNHE LIU, Cornell University, USA
NATE FOSTER, Cornell University, USA
TOBIAS KAPPÉ, Open University, the Netherlands and ILLC, University of Amsterdam, the Netherlands
DEXTER KOZEN, Cornell University, USA

Early programming models for software-defined networking (SDN) focused on basic features for controlling network-wide forwarding paths, but more recent work has considered richer features, such as packet scheduling and queueing, that affect performance. In particular, *PIFO trees*, proposed by Sivaraman et al., offer a flexible and efficient primitive for *programmable* packet scheduling. Prior work has shown that PIFO trees can express a wide range of practical algorithms including strict priority, weighted fair queueing, and hierarchical schemes. However, the semantic properties of PIFO trees are not well understood.

This paper studies PIFO trees from a programming language perspective. We formalize the syntax and semantics of PIFO trees in an operational model that decouples the scheduling policy running on a tree from the topology of the tree. Building on this formalization, we develop compilation algorithms that allow the behavior of a PIFO tree written against one topology to be realized using a tree with a different topology. Such a compiler could be used to optimize an implementation of PIFO trees, or realize a logical PIFO tree on a target with a fixed topology baked into the hardware. To support experimentation, we develop a software simulator for PIFO trees, and we present case studies illustrating its behavior on standard and custom algorithms.

CCS Concepts: • **Theory of computation → Semantics and reasoning**; • **Networks** → *Network properties.*

Additional Key Words and Phrases: packet scheduling, formal semantics, programmable scheduling

ACM Reference Format:
Anshuman Mohan, Yunhe Liu, Nate Foster, Tobias Kappé, and Dexter Kozen. 2023. Formal Abstractions for Packet Scheduling. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 269 (October 2023), 25 pages. https://doi.org/10.1145/3622845

1 INTRODUCTION

Over the past decade, programmable networks have gone from a dream to reality [Foster et al. 2020]. But why do network owners want to program the network? Although there has been some buzz around trendy topics like self-driving networks [Liu et al. 2020] and in-network computing [Dang et al. 2015; Jin et al. 2018, 2017], network owners often have less flashy priorities: they simply want to build networks that provide reliable service under uncertain operating conditions. Doing this well in practice requires not just fine-grained control over routing, but also prioritizing certain packets over others—i.e., controlling packet scheduling—using algorithms that operate at line rate.

Today, most routers support just a few scheduling algorithms—e.g., strict priority, weighted fair queueing, etc.—that are baked into the hardware. An administrator can select from these algorithms

Authors' addresses: Anshuman Mohan, amohan@cs.cornell.edu, Cornell University, USA; Yunhe Liu, yunheliu@cs.cornell.edu, Cornell University, USA; Nate Foster, jnfoster@cs.cornell.edu, Cornell University, USA; Tobias Kappé, t.kappe@uva.nl, Open University, the Netherlands and ILLC, University of Amsterdam, the Netherlands; Dexter Kozen, kozen@cs.cornell.edu, Cornell University, USA.

This work is licensed under a Creative Commons Attribution-NoDerivatives 4.0 International License.
© 2023 Copyright held by the owner/author(s).

2475-1421/2023/10-ART269
https://doi.org/10.1145/3622845

and configure their parameters to some extent, but they typically cannot implement entirely new algorithms. To get around this, Sivaraman et al. [2016b] proposed a new model for *programmable* packet schedulers based on a novel data structure called a *PIFO tree*.[1] This relatively simple data structure can be instantiated to not only realize a wide range of well-studied packet scheduling algorithms, but also compose them hierarchically. It seems likely that PIFO trees will be supported on network devices in the near future—indeed, the original paper on PIFO trees presented a detailed hardware design and demonstrated its feasibility, and researchers have also started to explore how the PIFO abstraction can be emulated on fixed hardware [Alcoz et al. 2020; Vass et al. 2022].

Informally, a PIFO tree associates a PIFO with each of its nodes. The leaf PIFOs hold buffered packets and the internal PIFOs hold scheduling data. To insert a packet into a PIFO tree, we first determine the leaf where the packet should be buffered, and then walk down the tree from the root to that leaf. At each internal node along this path, we insert into the node's PIFO a downward reference to the next node. At the leaf node we insert the packet itself, and this completes the insertion. The subtlety comes from choosing the *rank*s with which downward references and packets are inserted into PIFOs; these ranks are determined by a program called a *scheduling transaction*. Conversely, releasing a packet employs a simple, fixed algorithm: pop the root's PIFO to get an index to its most favorably ranked child, and recurse on that child until arriving at a leaf. Popping the leaf gives us the tree's most favorably ranked packet, which is then emitted.

Despite their operational simplicity, relatively little is known about how the topology of PIFO trees affects the expressivity of their scheduling algorithms. We are also not aware of any work that studies how to translate scheduling algorithms written for a tree with one topology onto another tree with a different topology. From a theoretical perspective, this question is interesting because it offers insights into the expressiveness of PIFO trees. From a practical perspective, it is important because it allows chip designers to focus on developing high-speed implementations with fixed topologies, freeing up hardware resources for implementing the scheduling logic.

In this paper we develop the first formal account of PIFO trees, using tools and techniques from the field of programming languages to model their semantics. We then embark on a comprehensive study of PIFO trees, examining how expressiveness is affected by variations in topology. This leads to our main result: an algorithm that can compile a scheduling policy written against one PIFO tree onto another PIFO tree of a different topology. We furthermore develop a reference implementation of our algorithm, as well as a simulator that we use to validate the algorithm. Finally, we compare the behavior of the simulator against scheduling algorithms running on a state-of-the-art switch.

More concretely, this paper makes the following contributions:

- We present the first formal syntax and semantics for PIFO trees, emphasizing the separation of concerns between the topology of a tree and the scheduling algorithm running on it.
- We study the semantics of PIFO trees in terms of the permutations they produce, and use this as a tool to formally distinguish the expressiveness of PIFO trees based on their topologies.
- We propose a fast algorithm that compiles a PIFO tree into a regular-branching PIFO tree, and a slower algorithm that compiles a PIFO tree into an arbitrary-branching PIFO tree. Our algorithms are accompanied by formal proofs characterizing when compilation is possible.
- We provide an implementation of the first algorithm, as well as a simulator for PIFO trees. We use this simulator to validate that our compiler preserves PIFO tree behavior, and compare PIFO trees against scheduling algorithms implemented on a hardware switch.

Overall, this work takes a first step toward higher-level abstractions for specifying scheduling algorithms by developing compilation tools, and suggests directions for future work on scheduling, in the networking domain and beyond.

[1] A PIFO is just a priority queue (*push-in-first-out*) that is additionally defined to break ties in *first-in-first-out* (*FIFO*) order.

2 OVERVIEW

We start by discussing what is perhaps the most obvious way to implement software-defined scheduling: a programmable priority queue. In this model, incoming packets are ranked using a (user-defined) *scheduling transaction*, and a packet is enqueued according to its rank. In general, the scheduling transaction may read and write state. To dequeue a packet, we simply remove the most favorably ranked packet (i.e., the one with the with lowest rank or, equivalently, highest priority).

Although such an approach is simple and relatively easy to implement, it is unable to express a basic feature of many packet scheduling algorithms: the ability to reorder buffered packets after they are enqueued. To address this shortcoming, we introduce PIFO trees [Sivaraman et al. 2016b], which extend the simple programmable queue model and support such reordering.

2.1 Programmable Priority Queues

Suppose you are responsible for programming a switch where incoming packets can be divided into two flows, coming to you from data centers in Rochester and Buffalo respectively. The flows are to be given equal priority up to the availability of packets. This last caveat is important: if one flow becomes inactive and the other stays active, then the active flow is given free rein to use the excess bandwidth until the other flow starts up again, at which point it again receives its fair share.

Fortunately, the queue in your network device is programmable: it allows you to specify a *scheduling transaction*, which assigns an integer rank to each packet and may also update some internal state. The packet is then inserted into a priority queue, or PIFO, which orders the packets by rank. Whenever the link becomes available, a different component in the network device removes the most favorably ranked packet from the queue and transmits it.

The scenario where we aim for an equal split between R and B traffic can be implemented by assigning ranks to incoming packets such that the contents of the queue interleave these flows whenever possible, and by maintaining some state. Specifically, our aim will be to maintain a queue that has one of the following three forms at any given time. Note that, in keeping with network queueing tradition, the most favorably ranked item in the queue is on the *right*, which we call the *head* of the queue. These queues are read from right to left, in the order they will be dequeued.

$$\mathsf{B}_n, \dots, \mathsf{B}_1,\ (\mathsf{R}, \mathsf{B})^* \qquad \mathsf{R}_n, \dots, \mathsf{R}_1,\ (\mathsf{R}, \mathsf{B})^* \qquad (\mathsf{R}, \mathsf{B})^*$$

Here, we write $(\mathsf{R}, \mathsf{B})^*$ to stand in for a balanced (possibly zero) number of interleaved R and B packets interleaved in either order, such as R, B, R, B or B, R, B, R, B, R.

Let us see how these three cases accept a new R or B packet. We just have a PIFO, so our power is limited: all we can do is assign the incoming packet a rank and then push the packet into the PIFO at that rank. The packet is inserted among the previously buffered packets without affecting the relative order of those packets. We distinguish based on the three possible shapes of the queue:

- To enqueue a B into $\mathsf{B}_n, \dots, \mathsf{B}_1,\ (\mathsf{R}, \mathsf{B})^*$, give it any rank that puts it within the list of unbalanced Bs. Making it B_{n+1} puts it on the far left and preserves arrival order within the B flow. To enqueue an R, put it just before or just after B_1 (depending on whether the item at the head of the queue is R or B). This extends the initial (balanced) part of the queue.
- To enqueue a B or an R into $\mathsf{R}_n, \dots, \mathsf{R}_1,\ (\mathsf{R}, \mathsf{B})^*$, act symmetrically to the previous case.
- To enqueue a B or an R into $(\mathsf{R}, \mathsf{B})^*$, assign it a rank that puts it on the very left. This creates a queue of either the first or the second kind.

2.2 Achieving Balance within Flows

The solution proposed for sharing bandwidth between R and B works well thus far, but now a complication arises. As it turns out, traffic from Rochester can be further divided by destination:

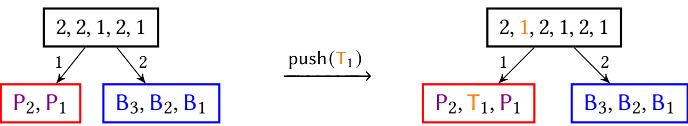


Fig. 1. A PIFO tree allows buffered packets to be reordered.

some packets are bound for Toronto and some for Pittsburgh. Traffic to these sites also needs to be balanced equally in a 1:1 split, but *within* the share of traffic allocated to R packets. That is, every two packets released should contain one R and one B, and every two R packets released should contain one T and one P. Again, this is up to the availability of packets.

We now highlight a particular case that may arise when trying to program a scheduling transaction that adheres to these constraints, and show that the desired split cannot be achieved using our single programmable queue. Suppose T is initially silent, while P and B are active. In this case, our scheduler should balance P and B *evenly*: in the absence of T traffic, P traffic gets to take up the entire share allocated to R traffic. Under this policy, the queue might take the following form:

$$B_3, B_2, P_2, B_1, P_1$$

So far, so good: opportunities for transmission are divided evenly between P and B, up to availability of packets. But now a Toronto-bound packet T_1 arrives, and our scheduling transaction needs to insert it somewhere in this queue. With an eye to the 1:1 split *within* the R flow, T_1 can go either before or after P_1 but it really does need to go before P_2—after all, if this is not the case, then opportunities for R traffic are not being split evenly between T and P. This leaves us with three possible choices for inserting T_1, which would result in one of the following queues:

$$B_3, B_2, P_2, T_1, B_1, P_1 \qquad B_3, B_2, P_2, B_1, T_1, P_1 \qquad B_3, B_2, P_2, B_1, P_1, T_1$$

However, *all* of these choices violate the 1:1 contract between R and B—recall that both T and P packets are still R packets, so the options above all propose to send two Rs in a row while an unbalanced B languishes in the back. Queues that *do* satisfy the balance requirements include

$$B_3, P_2, B_2, T_1, B_1, P_1 \qquad B_3, P_2, B_2, P_1, B_1, T_1$$

but note that these require the reordering of previously buffered packets relative to each other. Since our programmable queue model allows us to assign a rank only to the incoming packet, *it is impossible to achieve the desired behavior using a single PIFO.* Moreover, it was not obvious at the onset that the algorithm would require the reordering of previously buffered packets.

2.3 Introducing PIFO Trees

Sivaraman et al. [2016b] propose a remarkably elegant solution to the problem discussed above: instead of a single PIFO, use a *tree* of PIFOs. The leaves of this tree correspond to flows, and each leaf carries, in its PIFO, packets ranked according to rules local to the flow. Meanwhile, the internal nodes of the tree contain PIFOs that prioritize the opportunities for transmission *between classes of traffic*, without referring to a particular packet within those classes. This model provides a layer of indirection, which lets us eke out more expressiveness from the simple PIFO primitive while maintaining the flexibility of being able to program the relative priorities of each flow and subflow.

To illustrate how a PIFO tree operates, we now show how this model can be used to tackle the problem posed by the R/B/T/P traffic scenario discussed prior. The PIFO tree that we will use is depicted in Figure 1 on the left. Every node in this tree carries a PIFO, which we render, as before, with the most favorably ranked item on the right. An internal node carries transmission

opportunities for its children (here, indexed 1 and 2), while leaf nodes carry packets in their PIFOs. The tree on the left holds our original five packets, before the arrival of T_1. We maintain the left leaf as the R leaf and the right leaf as the B leaf; we show this with colored borders.

Dequeueing the tree on the left triggers two steps. First, we dequeue an element from the PIFO in the root; in this case we obtain 1, a reference to the left child. Second, we look at the left leaf, where dequeueing the PIFO returns the packet P_1. We emit this packet. In general, the pop operation recurses into a tree until it reaches a leaf, following the path guided by the references popped from the internal nodes. Repeating pop until the tree on the left is empty would yield the sequence B_3, B_2, P_2, B_1, P_1, meaning that the tree on the left matches the setup of our problematic example.

Now T_1 arrives. To push T_1 into the tree, we perform two steps:

(1) Since T_1 belongs to the R flow, we insert it into the PIFO of the left leaf. We must assign T_1 a rank; as noted before, T_1 needs to be ranked more favorably than P_2, but either position relative to P_1 is allowed. We place T_1 after P_1 in an attempt to stay closer to arrival order.
(2) We enqueue a 1 index (shown in orange because it was enqueued on account of a T packet) into the root node. We get to pick 1's rank before inserting it into the root PIFO; we choose a rank that maintains 1:1 harmony between the 1s and 2s.

The resulting tree is on the right in Figure 1. Note how repeatedly popping elements from this tree gives the desired ordering B_3, P_2, B_2, T_1, B_1, P_1. Critically, when the 1 at the root is popped, it causes the release of P_2 and not T_1. This is because the index 1 refers to an *opportunity* for the left child to propose a packet, not to any individual packet enqueued there. Indeed, the packet T_1 has already been released by the time 1 is dequeued. The release of T_1 was triggered by an older 1 index.

As before, we *just* used PIFOs—at enqueue, each item was assigned a rank and inserted without affecting the relative order of the older items in the PIFO—but the on-the-fly target switching that we have just seen, in which an index enqueued on account of one packet may eventually lead to the release of a different packet, is elegant and powerful. This is how PIFO trees facilitate the relative reordering of buffered packets despite the limitations of their relatively simplistic primitives.

2.4 Implementations and Expressiveness

A practical implementation of PIFO trees needs to operate at line rate—state-of-the-art devices achieve tens of terabits per second, which corresponds to 10 billion operations per second [Intel 2022]. Sivaraman et al. [2016b] proposed a hardware implementation, which overlays the PIFO tree on a mesh of interconnected hardware PIFOs, and a subset of this mesh is used to pass packets and metadata between parents and children on each operation. This is a very flexible approach, capable of accommodating more or less[2] any PIFO tree whose number of internal nodes and leaves does not exceed the number of hardware PIFOs. However, a fully connected mesh comes at a cost: it essentially requires the hardware to be configurable in terms of the connections that are used, which induces overhead in terms of performance, chip surface, cost, and complexity.

One could imagine a refinement of this hardware model, where the topology of the tree is fixed. This raises the question: *how much would a fixed topology of PIFO trees limit packet scheduling?* In this motivating section we have shown that a single PIFO (i.e., a PIFO tree consisting of just one leaf) is less expressive than a two-level PIFO tree, but it is not immediately clear how this distinction generalizes. Conversely, *is it possible for one PIFO tree to express the behavior of another, even if their topologies differ?* A constructive proof of such a correspondence would yield a compilation procedure, which could then be exploited to implement a user-designed PIFO tree on fixed hardware.

[2]The number of different scheduling policies to control each internal node is also limited; refer to op. cit. for further details.

2.5 Technical Contributions

Our aim is to undertake a comprehensive study of PIFO trees as a semantic model for programmable packet queues. Our technical contributions, and the remainder of this paper, are organized as follows.

- §3. We provide a rigorous model of the PIFO tree data structure, including its contents, operations, and well-formedness conditions.
- §4. Using this model, we formally prove that PIFO trees with n leaves are less expressive than PIFO trees with m leaves when $n < m$. By extension, this means that if one looks at the class of PIFO trees of degree k, taller PIFO trees are strictly more expressive.
- §5. We develop the notion of *(homomorphic) embedding* as a tool for showing that the scheduling behavior of one PIFO tree can be replicated by another.
- §6. We propose two procedures to compute embeddings between PIFO trees, if they exist. These algorithms map a packet scheduling architect's scheduling policy, written against whatever tree the architect finds intuitive, onto the fixed PIFO tree actually implemented in hardware.
- §7. We implement a simulator for PIFO tree behavior, as well as the algorithm from §6. We use this simulator to compare PIFO trees against their embeddings, as well as against standard algorithms implemented on a state-of-the-art programmable hardware switch.

3 STRUCTURE AND SEMANTICS

We now give a more formal definition of the syntax and semantics of PIFO trees. Let's fix some notation. When K is a set and $n \in \mathbb{N}$, we write K^n for the set of n-element lists over K. We denote such lists in the plural (e.g., $ks \in K^n$), write $|ks|$ for the list length n, read the i-th element (for $1 \leq i \leq n$) with $ks[i]$, and write $ks[k/i]$ for the list ks with the i-th element replaced by $k \in K$.

3.1 Structure

Since the topologies of PIFO trees are important for our results, we isolate them into a separate type, which we will momentarily use as a parameter. Conceptually, these are very straightforward: a tree topology is just a finite tree that does not hold any data.

Definition 3.1. The set of *(tree) topologies*, denoted Topo, is the smallest set satisfying the rules:

$$\frac{}{* \in \mathsf{Topo}} \qquad \frac{n \in \mathbb{N} \qquad ts \in \mathsf{Topo}^n}{\mathsf{Node}(ts) \in \mathsf{Topo}}$$

Here, $*$ is the topology of a single-node tree. Given a list ts of n topologies, $\mathsf{Node}(ts)$ is the topology of a node with ts as its children; the topology of the i-th child of $\mathsf{Node}(ts)$ is given by $ts[i]$.

Example 3.2. The topology of the PIFO tree from Figure 1 is given by $\mathsf{Node}(ts)$, where ts is a two-element list of topologies, with both entries equal to $*$.

As mentioned, PIFO trees are trees where each node holds a PIFO. A *leaf* node uses its PIFO to hold packets, while an *internal* node carries (1) a list of its children, and (2) a PIFO that holds valid references to those children. We can now formally define the set of PIFO trees of a given topology.

Definition 3.3. We fix an opaque set Pkt of *packets*, and a totally ordered set Rk of *ranks*. For any set S, we also presuppose a set $\mathsf{PIFO}(S)$ of PIFOs holding values from S and ordered by Rk.

The set of *PIFO trees* of a topology $t \in \mathsf{Topo}$, denoted $\mathsf{PIFOTree}(t)$, is defined inductively by

$$\frac{p \in \mathsf{PIFO}(\mathsf{Pkt})}{\mathsf{Leaf}(p) \in \mathsf{PIFOTree}(*)} \qquad \frac{n \in \mathbb{N} \qquad ts \in \mathsf{Topo}^n \qquad p \in \mathsf{PIFO}(\{1, \ldots, n\}) \qquad \forall 1 \leq i \leq n.\ qs[i] \in \mathsf{PIFOTree}(ts[i])}{\mathsf{Internal}(qs, p) \in \mathsf{PIFOTree}(\mathsf{Node}(ts))}$$

Here, $\mathsf{Leaf}(p)$ is a PIFO tree consisting of just one leaf holding the packet-PIFO p, and $\mathsf{Internal}(qs, p)$ is a PIFO tree whose root node holds p, an index-PIFO, as well as qs, a list of PIFO tree children.

3.2 Semantics

We can now define the partial function pop, which takes a PIFO tree and outputs a packet and an updated PIFO tree. We assume that PIFOs themselves support a partial function $\textsc{pop} : \mathsf{PIFO}(S) \rightharpoonup S \times \mathsf{PIFO}(S)$, which returns the most favorably ranked element of the PIFO and an updated PIFO.

Definition 3.4. For all topologies $t \in \mathsf{Topo}$, define $\mathrm{pop} : \mathsf{PIFOTree}(t) \rightharpoonup \mathsf{Pkt} \times \mathsf{PIFOTree}(t)$ by

$$\frac{\textsc{pop}(p) = (pkt, p')}{\mathrm{pop}(\mathsf{Leaf}(p)) = (pkt, \mathsf{Leaf}(p'))} \qquad \frac{\textsc{pop}(p) = (i, p') \qquad \mathrm{pop}(qs[i]) = (pkt, q')}{\mathrm{pop}(\mathsf{Internal}(qs, p)) = (pkt, \mathsf{Internal}(qs[q'/i], p'))}$$

In rules of this form, here and hereafter, the types of the variables are inferred from context. For example, p is a packet-PIFO in the rule on the left but an index-PIFO in the rule on the right.

When run on a leaf, pop simply applies $\textsc{pop}$ to its PIFO, returning the released packet and the updated node. When pop is run on an internal node, it applies $\textsc{pop}$ to its PIFO; the returned value is an index i pointing to a child node, on which pop is called recursively. The packet returned by the i-th child is also the present node's answer, along with a PIFO tree reflecting the effects of popping.

It is worth pointing out that pop is not a total function: it may be the case that $\mathrm{pop}(q)$ is undefined, most obviously when when q is a leaf or internal node with an empty PIFO (hence the call to $\textsc{pop}$ is undefined). This can also cascade up the tree, e.g., when an internal node's PIFO points to its i-th child, but the recursive pop call fails there. We expand on how to prevent this in Section 3.3.

To define the operation push, which inserts a packet into a PIFO tree, we will diverge slightly from Sivaraman et al. [2016b]. There, a PIFO tree chooses a leaf to enqueue the packet; the algorithm then walks from this leaf to the root and enqueues a downward reference at each node, where the rank of each enqueue is computed by a scheduling transaction attached to each non-root node. In our treatment, we assume that push is supplied with a *path* that contains all of this information precomputed; the path is constructed by an external *control*, which we will define in a moment.

Definition 3.5. The set of paths for a topology $t \in \mathsf{Topo}$, denoted $\mathsf{Path}(t)$, is defined as follows.

$$\frac{r \in \mathsf{Rk}}{r \in \mathsf{Path}(*)} \qquad \frac{ts \in \mathsf{Topo}^n \qquad 1 \leq i \leq n \qquad r \in \mathsf{Rk} \qquad pt \in \mathsf{Path}(ts[i])}{(i, r) :: pt \in \mathsf{Path}(\mathsf{Node}(ts))}$$

Intuitively, a path is a list $(i_1, r_1) :: \cdots :: (i_n, r_n) :: r_{n+1}$, where the first n elements (i_j, r_j) contain the index i_j of the next child and the rank r_j at which this index should be enqueued, and the last element r_{n+1} is the rank with which the packet should be enqueued in the leaf PIFO. For example, the path $(1, 10) :: (3, 5) :: 6$ means that the scheduling transaction wants us to perform three steps:

(1) Enqueue the index 1 in the root node with rank 10.
(2) Enqueue the index 3 in the first child of the root node with rank 5.
(3) Enqueue the packet itself in the third child of the first child of the root node with rank 6.

We now define the function push, which acts on a path for the topology of a PIFO tree. We assume each PIFO admits a function $\textsc{push} : \mathsf{PIFO}(S) \times S \times \mathsf{Rk} \to \mathsf{PIFO}(S)$, which takes a PIFO, an element, and a rank, and returns an updated PIFO with the element enqueued at the given rank.

Definition 3.6. Let $t \in \mathsf{Topo}$. For all $pkt \in \mathsf{Pkt}$, we define the function $\mathrm{push} : \mathsf{PIFOTree}(t) \times \mathsf{Pkt} \times \mathsf{Path}(t) \to \mathsf{PIFOTree}(t)$ inductively, as follows (types are inferred from context, as before).

$$\frac{\textsc{push}(p, pkt, r) = p'}{\mathrm{push}(\mathsf{Leaf}(p), pkt, r) = \mathsf{Leaf}(p')} \qquad \frac{\mathrm{push}(qs[i], pkt, pt) = q' \qquad \textsc{push}(p, i, r) = p'}{\mathrm{push}(\mathsf{Internal}(qs, p), pkt, (i, r) :: pt) = \mathsf{Internal}(qs[q'/i], p')}$$

When the node is a leaf, push simply enqueues the packet into the PIFO using the given rank. When the node is internal, it enqueues an index into the appropriate child into its own PIFO, and recurses by calling push on the appropriate child with an appropriately shortened path (in either order). If push succeeds on the child, an altered version of the child is returned; we functionally update the present node to reflect its new index-PIFO and list of PIFO tree children.

We model the scheduling algorithm that is run on the tree separately in a *control*, which keeps track of (1) a state from a fixed set St of states, as well as (2) a *scheduling transaction* that decides on a path (of the right topology) for each incoming packet, while possibly updating the state.

Definition 3.7. Let $t \in \text{Topo}$. A *control* over t is a triple (s, q, z), where $s \in \text{St}$ is the *current state*, q is a PIFO tree of topology t, and $z : \text{St} \times \text{Pkt} \to \mathsf{Path}(t) \times \text{St}$ is a function called the *scheduling transaction*. We write $\mathsf{Control}(t)$ for the set of controls over t.

Remark 1. To recover the presentation of the node-bound scheduling transactions from Sivaraman et al. [2016b], one can just project the return value of z to obtain a rank and child index for each node (updating the state only in the scheduling transaction for the root node). Conversely, individual scheduling transactions for each node can be glued together into a monolithic one that has the same effect. For the sake of brevity, we do not expand on these constructions.

3.3 Well-Formedness

From the development preceding, it should be clear that the pop and push operations do not change a PIFO tree's topology. There is, however, one more important invariant which these operations maintain, but which we have not yet discussed. This has to do with the *validity* of references carried by the internal nodes. Put simply, the pop and push operations maintain that if the PIFO of an internal node carries n references to its m-th child, then the leaves below that child carry exactly n packets. This prevents pop from "getting stuck" as it traverses a non-empty tree looking for a packet. We now formalize this notion in a typing relation, as follows.

Definition 3.8. Let S be a set, and $p \in \mathsf{PIFO}(S)$. We write $|p|$ for the size of p, which is the number of elements it holds; furthermore, when $s \in S$, we write $|p|_s$ for the number of times s occurs in p.

Let q be a PIFO tree. We write $|q|$ for the *size* of q, which is defined as the number of packets enqueued at the leaves. Formally, $|\cdot| : \mathsf{PIFOTree}(t) \to \mathbb{N}$ is defined for every $t \in \text{Topo}$ by

$$|\mathsf{Leaf}(p)| = |p| \qquad |\mathsf{Internal}(qs, p)| = |qs[1]| + \cdots + |qs[n]| \quad (n = |qs|)$$

We say that q is *well-formed*, denoted $\vdash q$, if it adheres to the following rules.

$$\frac{}{\vdash \mathsf{Leaf}(p)} \qquad \frac{\forall 1 \leq i \leq |qs|.\ \vdash qs[i] \wedge |p|_i = |qs[i]|}{\vdash \mathsf{Internal}(qs, p)}$$

Intuitively, leaves are well-formed, and an internal node is well-formed if, for all legal values of child-index i, the i-th child is itself a well-formed PIFO tree, and the number of times i occurs in the index-PIFO p is equal to the number of packets held by the i-th child.

With all of these notions in place, we are ready to formally state the invariants for PIFO trees.

Lemma 3.9. *Let $t \in \text{Topo}$ and $q \in \mathsf{PIFOTree}(t)$ and $pkt \in \text{Pkt}$. The following hold.*

(i) If $pt \in \mathsf{Path}(t)$, then $\mathsf{push}(q, pkt, pt)$ is well-defined, and $\vdash \mathsf{push}(q, pkt, pt)$.
(ii) If $|q| > 0$ and $\vdash q$, then $\mathsf{pop}(q)$ is well-defined, and $\vdash q'$ where $\mathsf{pop}(q) = (pkt, q')$.

Proof sketch. Both of these follow by (dependent) induction on t. □

4 LIMITS TO EXPRESSIVENESS

We have already suggested that some PIFO trees can express more policies than others. This section shows formally that, in general, PIFO trees with more leaves are more expressive. In turn, this tells us that a fixed-topology hardware implementation of PIFO trees needs to support a sufficient number of leaves if it is to express policies of practical interest.

Throughout this section, we use word notation for finite sequences of varying length, to contrast with the list notation used for sequences of fixed length. For instance, we will use juxtaposition to construct lists of packets $pkt_3 pkt_2 pkt_1$ and write $\cdot$ for the concatenation operator. The constant ϵ denotes an empty sequence, and S^* is the set of all finite sequences over elements of a set S. For the sake of brevity, most proofs are deferred to the extended version [Mohan et al. 2023b, Appendix A].

4.1 Simulation

Up to this point, we have been imprecise about what it means for one PIFO tree to replicate the behavior of another, but now we need a formal notion. To this end, we instantiate a tried and true idea from process algebra [Milner 1971], and more generally, coalgebra [Rutten 2000].

Intuitively, we wish to ask whether, given PIFO trees q_1 and q_2, any pop (resp. push) performed on q_1 can be mimicked by a pop (resp. push) on q_2, such that q_1 and q_2 schedule packets identically.

Definition 4.1. Let $t_1, t_2 \in \mathsf{Topo}$. We call a relation $R \subseteq \mathsf{PIFOTree}(t_1) \times \mathsf{PIFOTree}(t_2)$ a *simulation* if it satisfies the following conditions, for all $pkt \in \mathsf{Pkt}$ and $q_1 \; R \; q_2$:

(1) If $\mathsf{pop}(q_1)$ is undefined, then so is $\mathsf{pop}(q_2)$.
(2) If $\mathsf{pop}(q_1) = (pkt, q_1')$, then $\mathsf{pop}(q_2) = (pkt, q_2')$ such that $q_1' \; R \; q_2'$.
(3) For all $pt_1 \in \mathsf{Path}(t_1)$, there exists a $pt_2 \in \mathsf{Path}(t_2)$ such that

$$\mathsf{push}(q_1, pkt, pt_2) \;\; R \;\; \mathsf{push}(q_2, pkt, pt_2).$$

If such a simulation exists, we say that q_1 *is simulated by* q_2, and we write $q_1 \preceq q_2$.

It follows that, if c_1 is a control for q_1, then there exists a control c_2 for q_2 that can make q_2 behave in the same way as q_1. Conversely, if q_1 is *not* simulated by q_2, then there exists some control c_1 for q_1 whose scheduling decisions cannot be replicated by *any* control for q_2.

Example 4.2. To build intuition, let us construct a simple simulation between two trees of the same topology. Let $t \in \mathsf{Topo}$, and let $s : \mathsf{Rnk} \longrightarrow \mathsf{Rnk}$ be some monotone function on Rnk. We define $R_s \subseteq \mathsf{PIFOTree}(t) \times \mathsf{PIFOTree}(t)$ as the relation where $q_1 \; R_s \; q_2$ if and only if q_1 and q_2 agree on the contents of PIFOs in corresponding (leaf or internal) nodes, except that if r is the rank of an item in q_1, then $s(r)$ is the rank of that item in q_2. It is then easy to show that R_s is a simulation; in particular, when $pkt \in \mathsf{Pkt}$ and $pt_1 \in \mathsf{Path}(t)$ such that $\mathsf{push}(q_1, pkt, pt_1) = q_1'$, we can apply s to all the ranks in pt_1 to obtain $pt_2 \in \mathsf{Path}(t)$, and note that $q_1' \;\; R_s \;\; \mathsf{push}(q_2, pkt, pt_2)$.

4.2 Snapshots and Flushes

Showing that a relation on PIFO trees is a simulation is not always trivial, but typically doable. In contrast, showing that a PIFO tree q_1 *cannot* be simulated by a PIFO tree q_2 is a different endeavor altogether. To argue this formally, we have to find a property that remains true for q_1 over a sequence of actions, but is eventually falsified when mimicking those actions on q_2. We now introduce *snapshots* and *flushing* as tools to formalize such invariants, and show how the two relate.

A snapshot tells us about the relative order of packets already fixed by leaf nodes alone, but completely disregards the ordering dictated by the indices in the tree's internal PIFOs.

Definition 4.3. Let $p \in \mathsf{PIFO}(S)$ be a PIFO over some set S. We write $\textsc{flush}(p)$ for the sequence of elements (from S^*) retrieved by repeatedly calling $\textsc{pop}$ on the PIFO p until it is empty, with the

last POPped element on the far left. Furthermore, let q be a PIFO tree. We inductively define the *snapshot* of q, denoted $\mathsf{snap}(q)$, as a sequence of sequences given by:

$$\mathsf{snap}(\mathsf{Leaf}(p)) = [\textsc{flush}(p)]$$
$$\mathsf{snap}(\mathsf{Internal}(qs, p)) = \mathsf{snap}(qs[1]) +\!\!+ \cdots +\!\!+ \mathsf{snap}(qs[n]) \quad (n = |q|)$$

To prevent confusion between the different levels of finite sequences, we write $[x]$ to denote a single-element list containing x, and denote list concatenation on the upper level by $+\!\!+$.

A more delicate concept, flush, *does* take into account the indices enqueued in the tree's internal PIFOs: it yields the sequence obtained by popping a well-formed PIFO tree until it is empty. Keeping with convention, flush puts the most favorably ranked packet in the PIFO at the right of its output.

Definition 4.4. Let $t \in \mathsf{Topo}$ and let $q \in \mathsf{PIFOTree}(t)$ be a well-formed PIFO tree. We define $\mathsf{flush}(q)$ as a list of packets by induction on $|q|$, as follows:

$$\frac{|q| = 0}{\mathsf{flush}(q) = \epsilon} \qquad \frac{|q| > 0 \qquad \mathsf{pop}(q) = (pkt, q')}{\mathsf{flush}(q) = \mathsf{flush}(q') \cdot pkt}$$

LEMMA 4.5. *If q is a well-formed PIFO tree,* $\mathsf{flush}(q)$ *is an interleaving of the lists in* $\mathsf{snap}(q)$.

It is possible for PIFO trees to disagree on their snapshots, but still be in a simulation—for instance, a PIFO tree could simulate a PIFO tree of the same topology, while permuting the contents of some of the leaves. However, well-formed PIFO trees that are in simulation must agree on flush.

LEMMA 4.6. *Let q_1 and q_2 be well-formed PIFO trees. If $q_1 \preceq q_2$, then* $\mathsf{flush}(q_1) = \mathsf{flush}(q_2)$.

4.3 Fewer Leaves are Less Expressive

We now put the formalisms introduced above to work, by constructing, for each topology with n leaves, a PIFO tree that *cannot* be simulated by any PIFO tree with fewer leaves. We show that packets in different leaves can always be permuted by a specially chosen sequence of actions, while packets that appear in the same leaf are forced to obey the relative order they have in that leaf.

Definition 4.7. Let $t \in \mathsf{Topo}$. We write $|t|$ for the *number of leaves* in t; formally,

$$|*| = 1 \qquad\qquad |\mathsf{Node}(ts)| = |ts[1]| + \cdots + |ts[n]| \quad (n = |ts|)$$

Lemma 4.5 tells us that the flush of a PIFO tree is an interleaving of the lists in its snapshot. In particular, when each leaf holds a single element, *any* ordering of the packets in this tree is an interleaving of these snapshots. Say each leaf indeed holds only one packet. Can any ordering of packets be achieved? The following lemma answers this question in the positive, and in the process provides us with a construction to obtain a tree that describes any permutation.

LEMMA 4.8. *Let $t \in \mathsf{Topo}$, and let $P = \{pkt_1, \ldots, pkt_{|t|}\}$ be a set of distinct packets. For each permutation π on P, there exists a well-formed PIFO tree $q_\pi \in \mathsf{PIFOTree}(t)$ such that*

$$\mathsf{flush}(q_\pi) = \pi(pkt_{|t|}) \cdots \pi(pkt_1) \qquad\qquad \mathsf{snap}(q_\pi) = [pkt_1, \ldots, pkt_{|t|}]$$

As it happens, PIFO trees as constructed in this way for a fixed topology are closely related, in the sense that each q_π can be transformed into each $q_{\pi'}$ (up to simulation) via a specific sequence of operations. The idea is to use push to append the contents of the PIFO of each internal node in $q_{\pi'}$ to the PIFO of the corresponding node in q_π. The packets pushed will be the dummy packets pkt_{X}, which will be scheduled at the head of each leaf. A sequence of pops will then clear these dummy packets, as well as the original contents of the internal node PIFOs. The tree that results is very similar to q_π, except that the ranks used in its internal PIFOs may have shifted by some constant.

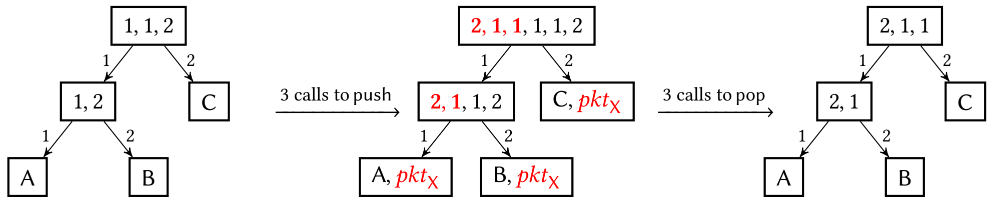


Fig. 2. Effecting permutations in $2|t|$ steps.

As an example, consider the PIFO tree on the left in Figure 2, which flushes to ABC. We can push pkt_X three times to obtain the PIFO tree in the middle, where the added contents are given in red. Calling pop three times yields the PIFO tree on the right, which itself flushes to CBA.

Lemma 4.9. *Let $t \in$ Topo, let $P = \{pkt_1, \ldots, pkt_{|t|}\} \subseteq$ Pkt, and let π, π' be permutations on P. Further, let pkt_X be a packet not occurring in P. There exists some $q \in \mathsf{PIFOTree}(t)$ and a sequence of $|t|$ pushes of pkt_X followed by $|t|$ pops that transforms q_π into q, such that $q \leq q_{\pi'}$.*

Remark 2. Lemmas 4.8 and 4.9 can be generalized to PIFO trees with multiple elements enqueued at each leaf. In a more lax notion of simulation, where the simulating tree may respond with more than one push or pop operation, this can be used to interrelate the semantics of PIFO trees with differing topologies. We refer to the extended version [Mohan et al. 2023b, Appendix B] for details.

With this transformation lemma in place, we are ready to formally state and prove a critical theorem: a PIFO tree can never be simulated by another PIFO tree with fewer leaves. Contrapositively, if a PIFO tree q_1 is simulated by a PIFO tree q_2, then q_2 has at least as many leaves as q_1.

Theorem 4.10. *Let $t_1, t_2 \in$ Topo with $|t_1| > |t_2|$. For all $q_1 \in \mathsf{PIFOTree}(t_1)$ and $q_2 \in \mathsf{PIFOTree}(t_2)$ such that $\vdash q_1$ and $\vdash q_2$, we have that $q_1 \leq q_2$ does* not *hold.*

Proof sketch. Take a set of distinct packets $P = \{pkt_1, \ldots, pkt_{|t_1|}\}$. We first consider the case where $q_1 = q_{\mathrm{id}}$, in which id is the identity permutation. If q_1 is simulated by $q_2 \in \mathsf{PIFOTree}(t_2)$, then q_2 has a leaf with at least two packets, pkt_i and pkt_j. By Lemma 4.9, we can then permute those packets in q_1 to obtain a PIFO tree q_1', using a sequence of push and pop operations that only involve a dummy packet pkt_X. Any way of applying push and pop operations to q_2 using this dummy packet will never change the position of pkt_i and pkt_j, because they are in the same leaf; hence, q_2 cannot simulate q_1. The more general case can be reduced to the above for well-formed trees, by simply popping q_1 until it is empty, and then pushing packets to turn it into q_{id}. □

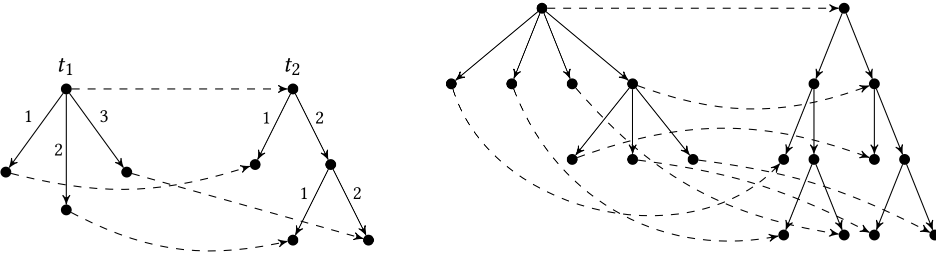
5 EMBEDDING AND SIMULATION

In the previous section, we formalized what it means for one tree to simulate the queuing behavior of another, and showed that the number of leaves is important: a PIFO tree *cannot* simulate a PIFO tree with more leaves. We now turn our attention to the converse: when can one PIFO tree simulate another? As explained, we will focus primarily on the *topology* of PIFO trees. Proofs of the lemmas stated here are available in the extended version of this paper [Mohan et al. 2023b, Appendix C].

In a nutshell, the results in this section tell us that a hardware implementation of PIFO trees may fix a certain topology. As long as the topologies of PIFO trees designed by the user embed in this topology (see below), the hardware implementation will be able to replicate their behavior.

5.1 Embedding

We start by developing a notion of *embedding* at the level of PIFO tree topologies, and show that if $t_1 \in$ Topo embeds inside $t_2 \in$ Topo, then PIFO trees over t_1 can be simulated by PIFO trees over t_2.



(a) Embedding a ternary tree into a binary tree.

(b) A more complex embedding.

Fig. 3. Two examples of embedding.

Intuitively, an embedding maps nodes of one topology to nodes of the other in a way that allows internal nodes to designate (new) intermediate nodes as their children. To make this precise, we need a way to refer to nodes and leaves internal to a topology, which we formalize as follows.

Definition 5.1 (Addresses). Let $t \in \mathsf{Topo}$. We write $\mathsf{Addr}(t)$ for the set of *(node) addresses* in t, which is defined as the smallest subset of $\mathbb{N}^*$ satisfying the rules

$$\frac{}{\epsilon \in \mathsf{Addr}(t)} \qquad \frac{n \in \mathbb{N} \qquad ts \in \mathsf{Topo}^n \qquad \alpha \in \mathsf{Addr}(ts[i]) \qquad 1 \leq i \leq n}{i \cdot \alpha \in \mathsf{Addr}(\mathsf{Node}(ts))}$$

Intuitively, ϵ addresses the root, while $i \cdot \alpha$ refers to the node pointed to by α in the i-th subtree.

When $\alpha \in \mathsf{Addr}(t)$, we write t/α for the subtree reached by α, defined inductively by

$$t/\epsilon = \epsilon \qquad \mathsf{Node}(ts)/(i \cdot \alpha) = ts[i]/\alpha$$

This defines a total function, because if $\alpha = i \cdot \alpha'$, then t is necessarily of the form $\mathsf{Node}(ts)$ for $ts \in \mathsf{Topo}^n$ such that $1 \leq i \leq n$, and $\alpha' \in \mathsf{Addr}(ts[i])$.

Note that this way of addressing nodes is compatible with the intuition of *ancestry*: if α and α' are addresses in t, then α points to an ancestor of the node referred to by α' precisely when α is a prefix of α'. This guides the definition of embedding, as follows.

Definition 5.2 (Embedding). Let $t_1, t_2 \in \mathsf{Topo}$. A *(homomorphic) embedding* of t_1 in t_2 is an injective map $f : \mathsf{Addr}(t_1) \to \mathsf{Addr}(t_2)$ such that, for all $\alpha, \alpha' \in \mathsf{Addr}(t_1)$, three things hold:

(1) f maps the root of t_1 to the root of t_2, i.e., if $\alpha = \epsilon$ then $f(\alpha) = \epsilon$.
(2) f maps leaves of t_1 to leaves of t_2, i.e., if $t_1/\alpha = *$, then $t_2/f(\alpha) = *$.
(3) f respects ancestry, i.e., α is a prefix of α' iff $f(\alpha)$ is a prefix of $f(\alpha')$.

For example, in Figure 3a, the ternary tree t_1 embeds inside the binary tree t_2 via the embedding f:

$$f(\epsilon) = \epsilon \qquad f(1) = 1 \qquad f(2) = 21 \qquad f(3) = 22$$

We do not explicate the embedding shown in Figure 3b, and we drop the child-indices to lighten the presentation, but observe that the three conditions of an embedding hold. In general, neither the source nor the target tree needs to be regular-branching so long as these conditions are obeyed.

Given an embedding of a topology t_1 into another topology t_2, we can obtain embeddings of subtrees of t_1 into subtrees of t_2; this gives us a way to superimpose the inductive structure of topologies onto embeddings, which we will rely on for the remainder of this section.

LEMMA 5.3. *Let $t_1, t_2 \in \mathsf{Topo}$, and let $f : \mathsf{Addr}(t_1) \to \mathsf{Addr}(t_2)$ be an embedding. The following hold: (1) if $t_1 = *$, then $t_2 = *$ as well; and (2) if $t_1 = \mathsf{Node}(ts_1)$, then there exists for $1 \leq i \leq |ts_1|$ an embedding f_i of t_1/i inside $t_2/f(i)$ satisfying $f(i \cdot \alpha) = f(i) \cdot f_i(\alpha)$.*

5.2 Lifting Embeddings

Next, we develop a way to lift these embeddings so that they range over PIFO trees. Specifically, given an embedding $f : \mathsf{Addr}(t_1) \to \mathsf{Addr}(t_2)$, we lift it into a map $\widehat{f} : \mathsf{PIFOTree}(t_1) \to \mathsf{PIFOTree}(t_2)$. Intuitively, if q is a PIFO tree over t_1, $\widehat{f}(q)$ is a PIFO tree over t_2 that simulates q. The map places packets at the leaves of the input PIFO tree at the corresponding leaves of the output PIFO tree, and populates the PIFOs at the internal nodes of the output tree in a way that preserves push and pop.

Definition 5.4. Let $t_1, t_2 \in \mathsf{Topo}$, and let f be an embedding of t_1 inside t_2. We lift f to a map $\widehat{f}$ from $\mathsf{PIFOTree}(t_1)$ to $\mathsf{PIFOTree}(t_2)$ by recursion on t_1.

- In the base, where we have $t_1 = *$, we choose $\widehat{f}(q) = q$. This is well-defined by Lemma 5.3.
- In the recursive step, let $t_1 = \mathsf{Node}(ts_1)$, $n = |ts_1|$, $q = \mathsf{Internal}(qs, p)$, $p \in \mathsf{PIFO}(\{1, \dots, n\})$, and, for $1 \le i \le n$, note that $qs[i] \in \mathsf{PIFOTree}(ts_1[i])$. For each prefix α of $f(i)$ for some $1 \le i \le n$, we construct $\widehat{f}(q)_\alpha \in \mathsf{PIFOTree}(t_2/\alpha)$, by an inner *inverse* recursion on α, starting from $f(i)$ for $1 \le i \le n$ and working our way to ϵ. This eventually yields $\widehat{f}(q) = \widehat{f}(q)_\epsilon \in \mathsf{PIFOTree}(t_2/\epsilon) = \mathsf{PIFOTree}(t_2)$:
  - In the inner base, $\alpha = f(i)$ for some $1 \le i \le n$. We choose $\widehat{f}(q)_\alpha = \widehat{f_i}(qs[i])$, where f_i embeds t_1/i inside $t_2/f(i)$, as obtained through Lemma 5.3. This is well-defined because f is injective, and because $qs[i] \in \mathsf{PIFOTree}(ts_1/i)$, as ts_1/i is a subtree of t_1.
  - In the inner recursive step, α points to an internal node of t_2 with, say, m children. For all $1 \le j \le m$, $\alpha \cdot j$ is a prefix of some $f(i)$. By recursion, we know $\widehat{f}(q)_{\alpha \cdot j} \in \mathsf{PIFOTree}(t_2/(\alpha \cdot j))$ exists. We create a new PIFO p_α by replacing the indices i in p (found at the root of q, see above) by $1 \le j \le m$ such that $\alpha \cdot j$ is a prefix of $f(i)$ if such a j exists, and removing them otherwise. Finally, we choose: $\forall 1 \le i \le m.\ qs_{f,\alpha}[j] = \widehat{f}(q_{\alpha \cdot j})$ and $\widehat{f}(q)_\alpha = \mathsf{Internal}(qs_{f,\alpha}, p_\alpha)$. By construction, we then have that $\widehat{f}(q)_\alpha \in \mathsf{PIFOTree}(t_2/\alpha)$.

To build intuition, let us revisit the embedding f we showed in Figure 3a and lift it to operate on PIFO trees. The upper row of Figure 4 shows this lifting at the point when we are computing $\widehat{f}$ for the root node, and we already have at hand the liftings of the form $\widehat{f_i}(s_i)$ for the root's children s_i. We present this in two steps. First, we remove indices of children that no longer exist below us ($\boxed{3, 1, 2, 2, 3, 1, 2}$ becomes $\boxed{3, 2, 2, 3, 2}$ because we remove 1s). Second, we renumber indices for the new address regime ($\boxed{3, 1, 2, 2, 3, 1, 2}$ becomes $\boxed{2, 1, 2, 2, 2, 1, 2}$ and $\boxed{3, 2, 2, 3, 2}$ becomes $\boxed{2, 1, 1, 2, 1}$).

As a sanity check, we verify that a lifted embedding preserves well-formedness of PIFO trees. The proof of the following property also serves as a small model for the proofs about lifted embeddings that follow, which all proceed by an induction that matches the recursive structure above.

Lemma 5.5. *Let $t_1, t_2 \in \mathsf{Topo}$, and suppose f embeds t_1 inside t_2. If $\vdash q$, then $\vdash \widehat{f}(q)$.*

Proof sketch. By induction on t_1. In the base, where $t_1 = *$, this holds because $\widehat{f}(q) = q$. Next, let $\mathsf{Node}(ts_1)$ and $n = |ts_1|$ and perform inverse induction on the prefixes of $f(i)$ for $1 \le i \le n$. Show, more generally, that (1) if α is such a prefix, then $\vdash \widehat{f}(q)_\alpha$, and (2) $|\widehat{f}(q)_\alpha|$ is equal to the sum of $|p|_i$ where $1 \le i \le n$ and α is a prefix of $f(i)$. Instantiating (1) with $\alpha = \epsilon$ implies the claim. □

5.3 Preserving pop

Simulation requires that the simulating tree can always be popped when the simulated tree can be popped. If q is well-formed and not empty, this must also be the case for $\widehat{f}(q)$—and hence if q can be popped so can $\widehat{f}(q)$ in this case. In fact, this is true regardless of well-formedness.

Lemma 5.6. *Let $t_1, t_2 \in \mathsf{Topo}$, and let f embed t_1 inside t_2. If $\mathsf{pop}(q)$ is defined, then so is $\mathsf{pop}(\widehat{f}(q))$.*

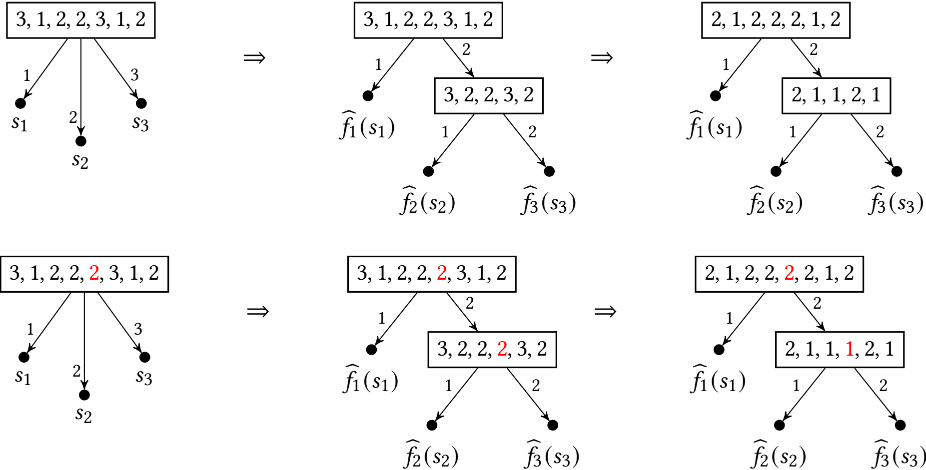


Fig. 4. Lifting an embedding (above), and preserving push across a lifted embedding (below). The source and target topologies as shown in Figure 3a. The new indices inserted are in red.

Proof sketch. By induction on t_1. In the base, where $t_1 = *$, the claim is trivial because $\widehat{f}(q) = q$. For the inductive step, let $t_1 = \mathsf{Node}(ts_1)$ and $n = |ts_1|$. Write $q = \mathsf{Internal}(qs, p)$. Since $\mathsf{pop}(q)$ is defined, p cannot be empty; write i for the index at the head of p. We then prove that $\mathsf{pop}(\widehat{f}(q)_\alpha)$ is defined for all prefixes α of $f(i)$ by inverse induction on α; when $\alpha = \epsilon$, this implies the claim. □

We furthermore need to show that if popping q yields a packet pkt and a new tree q', then popping $\widehat{f}(q)$ gives us the same packet pkt, and a tree q'' that simulates q'. To this end, we show something stronger, namely that pop *commutes* with $\widehat{f}(q)$, in the following sense.

Lemma 5.7. *Let $t_1, t_2 \in \mathsf{Topo}$, and let f be an embedding of t_1 inside t_2. Now* pop *is compatible with $\widehat{f}$, i.e., if* $\mathsf{pop}(q) = (pkt, q')$, *then* $\mathsf{pop}(\widehat{f}(q)) = (pkt, \widehat{f}(q'))$.

Proof sketch. By induction on t_1. In the base, where $t_1 = *$, the claim holds because $\widehat{f}(q) = q$. For the inductive step, let $t_1 = \mathsf{Node}(ts_1)$ and $n = |ts_1|$. Furthermore, let i be the index at the head of the PIFO attached to the root of q. As before, we proceed by inverse induction on the prefixes α of $f(j)$ with $1 \leq j \leq n$, arguing more generally that the following hold:

(1) If α is a prefix of $f(i)$, then $\mathsf{pop}(\widehat{f}(q)_\alpha) = (pkt, \widehat{f}(q')_\alpha)$.
(2) Otherwise, if α is not a prefix of $f(i)$, then $\widehat{f}(q)_\alpha = \widehat{f}(q')_\alpha$.

Instantiating the first property for the root address $\epsilon \in \mathsf{Path}(t_1)$ then tells us that:

$$\mathsf{pop}(\widehat{f}(q)) = \mathsf{pop}(\widehat{f}(q)_\epsilon) = (pkt, \widehat{f}(q')_\epsilon) = (pkt, \widehat{f}(q')).$$ □

Remark 3. For any $t \in \mathsf{Topo}$, we can define an operator that takes two PIFO trees $q_1, q_2 \in \mathsf{PIFOTree}(t)$ and produces a new PIFO tree $q_1; q_2 \in \mathsf{PIFOTree}(t)$, where the contents of the PIFO at each node in q_2 are concatenated to the corresponding node in q_1 (by shifting the weights of the former and inserting them). This operator is associative, and has the empty PIFO tree as its neutral element on both sides, which makes $\mathsf{PIFOTree}(t)$ a monoid. When restricted to well-formed PIFO trees, it is a *free monoid*: every PIFO tree can uniquely be written as the concatenation of PIFO trees where each PIFO holds at most one element. Further, given an embedding f of t_1 into t_2,

$\widehat{f}$ is a monoid homomorphism from $\mathsf{PIFOTree}(t_1)$ to $\mathsf{PIFOTree}(t_2)$. These properties can then be exploited to obtain an abstract proof of Lemma 5.7. Here we have chosen to give an explicit proof.

5.4 Preserving push

We continue by showing that $\widehat{f}$ is also compatible with push. To this end, we must show that, if we push the packet pkt into q with path pt, then there exists a path pt' such that pushing pkt into $\widehat{f}(q)$ with path pt' results in a tree that simulates $\mathsf{push}(q, pkt, pt)$. As it turns out, the correspondence between paths in t_1 and paths in t_2 can *also* be obtained from f—intuitively, it extends paths that lead to a leaf of t_1 to the corresponding leaf in t_2, while duplicating the ranks as necessary.

For example, let us revisit the topologies t_1 and t_2 from Figure 3a and push a packet into a PIFO tree with topology t_1 using the path $(2, 5) :: 7$. To preserve push on a tree with topology t_2, the path is translated into $(2, 5) :: (1, 5) :: 7$ by the embedding from the same figure. See the lower row of Figure 4 for an illustration. We give a formal definition of this translation below.

Definition 5.8. Let $t_1, t_2 \in \mathsf{Topo}$ and let f be an embedding of t_1 inside t_2. We define $\widetilde{f} : \mathsf{Path}(t_1) \to \mathsf{Path}(t_2)$ by induction on t_1. First, if $t_1 = *$, then every $pt \in \mathsf{Path}(t_1)$ is of the form r for $r \in \mathsf{Rnk}$; we set $\widetilde{f}(r) = r$. Since $t_2 = *$ by Lemma 5.3, this is well-defined.

Otherwise, if $t_1 = \mathsf{Node}(ts_1)$ with $|ts_1| = n$, then for $1 \leq i \leq n$ let f_i be the embedding of t_1/i into $t_2/f(i)$. Every element of $\mathsf{Path}(t_1)$ is of the form $(i, r) :: pt$ where $1 \leq i \leq n$, $r \in \mathsf{Rk}$ and $pt \in \mathsf{Path}(t_1/i)$. For every prefix α of $f(i)$, we define $\widetilde{f}((i, r) :: pt)_\alpha$ by inverse recursion. In the base, where $\alpha = f(i)$, we set $\widetilde{f}((i, r) :: pt)_\alpha = \widetilde{f_i}(pt)$. In the inductive step, where $\alpha = \alpha' \cdot j$, we set $\widetilde{f}((i, r) :: pt)_\alpha = (j, r) :: \widetilde{f}((i, r) :: pt)_{\alpha'}$. Finally, we define $\widetilde{f}((i, r) :: pt) = \widetilde{f}((i, r) :: pt)_\epsilon$.

We can now show that $\widehat{f}$ commutes with push if we translate the insertion path according to $\widetilde{f}$:

Lemma 5.9. *Let $t_1, t_2 \in \mathsf{Topo}$, $pkt \in \mathsf{Pkt}$, $pt \in \mathsf{Path}(t_1)$ and $q \in \mathsf{PIFOTree}(t_1)$. We have*

$$\widehat{f}(\mathsf{push}(q, pkt, pt)) = \mathsf{push}(\widehat{f}(q), pkt, \widetilde{f}(pt)).$$

Proof sketch. By induction on t_1. In the base, where $t_1 = *$, we know $\widehat{f}(q) = q$ and $\widetilde{f}(pt) = pt$, and further, since $\mathsf{push}(q, pkt, pt) \in \mathsf{PIFOTree}(t_1)$, we know $\widehat{f}(\mathsf{push}(q, pkt, pt)) = \mathsf{push}(q, pkt, pt)$. The claim then holds trivially. For the inductive step, let $t_1 = \mathsf{Node}(ts_1)$ and $n = |ts_1|$. We can then write $pt = (i, r) :: pt'$ where $pt' \in \mathsf{Path}(ts_1[i])$. Let $\alpha \in \mathsf{Addr}(t_1)$ be a prefix of some $f(j)$ with $1 \leq j \leq n$. We can then prove by inverse induction that the following properties hold:

(1) If α is a prefix of $f(i)$, then $\widehat{f}(\mathsf{push}(q, pkt, pt))_\alpha = \mathsf{push}(\widehat{f}(q)_\alpha, pkt, \widetilde{f}(pt)_\alpha)$.
(2) Otherwise, if α is not a prefix of $f(i)$, then $\widehat{f}(\mathsf{push}(q, pkt, pt))_\alpha = \widehat{f}(q)_\alpha$

When $\alpha = \epsilon$, the first property instantiates to the claim. □

5.5 Simulation

Now that we know that lifted embeddings are compatible with the push and pop operations on PIFO trees, we can formally state and prove the correctness of this operation as follows.

Theorem 5.10. *Let $t_1, t_2 \in \mathsf{Topo}$, $q \in \mathsf{PIFOTree}(t_1)$. If f embeds t_1 in t_2, then $q \preceq \widehat{f}(q)$.*

Proof. It suffices to prove that $R = \{(q', f(q')) : q' \in \mathsf{PIFOTree}(t_1)\}$ is a simulation.

(1) If $\mathsf{pop}(q')$ is defined, then so is $\mathsf{pop}(\widehat{f}(q'))$ by Lemma 5.6.
(2) If $\mathsf{pop}(q') = (pkt, q'')$, then $\mathsf{pop}(\widehat{f}(q')) = (pkt, \widehat{f}(q''))$ by Lemma 5.7.
(3) If $\mathsf{push}(q', pkt, pt)$, then choose $pt' = \widetilde{f}(pt)$ to find by Lemma 5.9 that

$$\mathsf{push}(q', pkt, pt) \;\; R \;\; \widehat{f}(\mathsf{push}(q', pkt, pt)) = \mathsf{push}(\widehat{f}(q'), pkt, pt'). \qquad \square$$

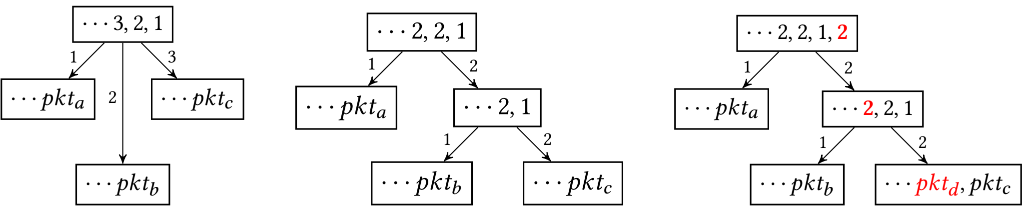


Fig. 5. Impossibility of simulation. The trees from left to right are q'_1, q'_2, q''_2. Although q'_1 ostensibly simulates q'_2, we can push pkt_d into q'_2 to get q''_2, and no push of pkt_d into q'_1 can simulate this.

Given an embedding f of $t_1 \in \mathsf{Topo}$ into $t_2 \in \mathsf{Topo}$, we can now translate a control $c = (s, q, z) \in \mathsf{Control}(t_1)$ into a control $c' = (s, \widehat{f}(q), z') \in \mathsf{Control}(t_2)$, where $z'(s, pkt) = (\widetilde{f}(pt), s')$ when $z(s, pkt) = (pt, s')$. Theorem 5.10 then tells us that c' behaves just like c; there is no overhead in terms of state, and the translation of the scheduling transaction z is straightforward.

Remark 4. Our results can be broadened to a more general model of scheduling. Say pop returned a path containing all of the nodes whose PIFOs were popped, along with their ranks, and that a control could react to a pop using a "descheduling transaction" that could look at this pop-path and update the state. This extended version of pop would still be compatible with embedding, i.e., Lemma 5.7 can be updated to show that if $\mathsf{pop}(q) = (pt, pkt, q')$, then $\mathsf{pop}(\widehat{f}(q)) = (\widetilde{f}(pt), pkt, \widehat{f}(q'))$.

5.6 A Counterexample

A natural question to ask next is whether embeddings can be lifted in the opposite way, i.e., if f embeds t_1 in t_2 and $q_2 \in \mathsf{PIFOTree}(t_2)$, can we use f to find a $q_1 \in \mathsf{PIFOTree}(t_1)$ such that $q_2 \preceq q_1$? Theorem 4.10 tells us that this is impossible if t_2 has more leaves than t_1. But even if t_1 embeds in t_2 and t_2 does not have any more leaves than t_1, such a mapping may be impossible to find.

Proposition 5.11. *There exist $t_1, t_2 \in \mathsf{Topo}$ s.t. for all PIFO trees $q_1 \in \mathsf{PIFOTree}(t_1)$ and $q_2 \in \mathsf{PIFOTree}(t_2)$, q_2 does not simulate q_1, even though t_1 embeds inside t_2 and $|t_1| = |t_2|$.*

Proof. Consider the embedding shown in Figure 3a, and say that t_1 and t_2 are as labeled in that figure. Consider any $q_1 \in \mathsf{PIFOTree}(t_1)$ and $q_2 \in \mathsf{PIFOTree}(t_2)$, and suppose towards a contradiction that q_1 simulates q_2. Take three distinct packets pkt_a, pkt_b, and pkt_c, and let q'_2 be the tree obtained from q_2 by pushing pkt_a, pkt_b, and pkt_c to the front of the first, second and third leaf respectively, such that pkt_a, pkt_b, and pkt_c will be popped in that order, as depicted in the center in Figure 5.

Because q_1 simulates q_2, we can push these packets to obtain a tree q'_1 that simulates q'_2. In this tree, pkt_a, pkt_b, and pkt_c *must* appear in different leaves—otherwise, we could execute a series of pushes and pops (as in Lemma 4.9) that changed the order of two packets in q'_2 but not in q'_1. Furthermore, because these packets are popped first from q'_2, they are also popped first from q'_1. We can therefore assume (without loss of generality) that q'_1 is as depicted on the left of Figure 5.

Now take a fourth packet pkt_d distinct from pkt_a, pkt_b and pkt_c, and push it into q'_2 to obtain the PIFO tree q''_2, depicted on the right in Figure 5 (new elements in red). In q''_2, the first four packets popped are pkt_b, pkt_a, pkt_c, and pkt_d. Because q'_1 simulates q'_2, there exists a way to push pkt_d into q'_1 that results in a PIFO tree q''_1 where the first four packets popped match those of q''_2. We can constrain the position of pkt_d in q''_1: pkt_d *must* be enqueued at the second position of the third leaf of q'_1—otherwise, if, for instance, pkt_d appeared in the same leaf as pkt_a in q''_1, then q''_2 could swap the relative order of pkt_d and pkt_a, but q''_1 could not (via the same technique as in Lemma 4.9).

Together, this means that, in q_1'', 3 must be enqueued near the head of the root PIFO. We have four possibilities for the root PIFO of q_1'', listed here along with the first four pops they would effect:

$3213 : pkt_c, pkt_a, pkt_b, pkt_d \qquad 3231 : pkt_a, pkt_c, pkt_b, pkt_d \qquad 3321/3321 : pkt_a, pkt_b, pkt_c, pkt_d$

Because none of these match the first four packets popped from q_2'', we have reached a contradiction. Our assumption that q_1 simulates q_2 must therefore be false, which proves the claim. □

6 EMBEDDING ALGORITHMS

So far, we showed that embeddings can be used to replicate the behavior of one PIFO tree with a PIFO tree of a different topology. To exploit these results in practice, we need to calculate an embedding of one (user-supplied) topology into another (hardware-mandated) topology. This section proposes two efficient algorithms that can be used to find such an embedding, if it exists.

6.1 Embedding into Complete d-ary Topologies

We start by treating the special case where the target of our embedding is a complete topology of fixed arity. In this case, the algorithm also has a favorable complexity.

THEOREM 6.1. *Let $t_1, t_2 \in \mathsf{Topo}$ such that t_2 is a complete d-ary topology. There is an $O(n \log n)$ algorithm ($n = |\mathsf{Addr}(t_1)|$) to determine whether t_1 embeds in t_2, and to find such an embedding if so.*

PROOF. We construct an embedding f of t_1 into complete d-ary topology of minimal height, in a greedy bottom-up fashion reminiscent of the construction of optimal Huffman codes [Cover and Thomas 2006]. Let $\alpha \in \mathsf{Addr}(t_1)$ and suppose, at some stage of the algorithm, we have for each $\alpha \cdot i \in \mathsf{Addr}(t_1)$ an embedding of $t_1/(\alpha \cdot i)$ inside a complete d-ary topology t_i^d of minimum height—in particular, when $\alpha \cdot i$ is a leaf, this is the trivial embedding. To get a minimum-height embedding of t_1/α inside a complete d-ary topology, we insert all of these topologies t_i^d into a min-priority queue, ranked by height. We repeat the following until the queue has one element:

(1) Extract up to d elements of the same minimum priority (say m).
(2) Create a new topology of height $m + 1$ with those elements as children. If desired, pad this new topology—add dummy leaves just below the root—to make it a *complete* d-ary topology.
(3) Insert this new topology with height $m + 1$ into the priority queue with priority $m + 1$.

Note, if there is only one topology of minimum height m, we do not need to form a new topology with one child. We simply reinsert the topology, unchanged, into the queue but with priority $m + 1$.

It follows from an exchange argument (given below; see [Kleinberg and Tardos 2006, Ch. 4] and [D'Angelo and Kozen 2023]) that the single remaining topology t is of the minimum height into which t_1/α embeds. An embedding of the whole topology is thus possible provided that t is no taller than t_2. The time required to process a node with d_s children is $O(d_s \log d_s)$ for the maintenance of the priority queue and $O(d_s)$ for all other operations, or $O(n \log n)$ in all, where $n = |t_1|$.

Here is the exchange argument. At any stage of the algorithm, let $t_1', \ldots, t_k'$ be a maximal set, up to size d, of queue elements with minimum priority m. Suppose the action of forming a new topology of height $m + 1$ with these children and inserting it into the queue was not the right thing to do, i.e., some other action would have led to a topology t' of optimal height. Assume without loss of generality that t_1' occurs deepest in t'; thus the path from the root to t_1' is the longest among the queue elements t_i'. Permute t' so that t_1' occurs rightmost. The topologies $t_2', \ldots, t_k'$ can be swapped for the siblings of t_1' in t', or, if t_1' has fewer than k siblings, added as a new child to the parent of t_1' in t', without increasing the height of the topology. Thus the action of the algorithm was correct.

For the case $k = 1$ with more than one topology left in the queue, the argument is the same, with the observation that in any embedding, t_1' must have a sibling of greater height. Therefore, it does not hurt to consider t_1' to be of height $m + 1$ even though its height is actually m. □

6.2 Embedding into Arbitrary Topologies

We now turn our attention to the more general case, where the target of embedding is an arbitrary PIFO topology. An embedding algorithm can still be achieved here, at the cost of a higher complexity.

THEOREM 6.2. *Let $t_1, t_2 \in$ Topo, such that each node in t_1 has at most d children. There is a polynomial-time algorithm to determine whether t_1 embeds in t_2, and to find such an embedding if so.*

PROOF. The algorithm is based on bottom-up dynamic programming. Construct a boolean-valued table T with entries $T(\alpha_1, \alpha_2)$ for $\alpha_1 \in \mathsf{Addr}(t_1)$ and $\alpha_2 \in \mathsf{Addr}(t_2)$ that says whether there exists an embedding that maps α_1 to α_2. Suppose that $\alpha_1 \in \mathsf{Addr}(t_1)$ has d_1 children, that we have determined the values of $T(\alpha_1', \alpha_2')$ such that α_1 (resp. α_2) is an ancestor (i.e., prefix) of α_1' (resp. α_2'), and that we wish to determine $T(\alpha_1, \alpha_2)$. The value will be true if we can find target nodes for the children of α_1 among the descendants of α_2 that form an antichain w.r.t. the ancestor/descendant relation.

Without loss of generality, we can ignore node α_2' as a candidate target for a child $\alpha_1 \cdot i$ if α_2' has a descendant α_2'' such that $T(\alpha_1 \cdot i, \alpha_2'') = \mathsf{true}$; any embedding that maps $\alpha_1 \cdot i$ to α_2' can map $\alpha_1 \cdot i$ to α_2'' instead. We now form a graph with nodes (α_1', α_2') such that $T(\alpha_1', \alpha_2') = \mathsf{true}$ and edges $((\alpha_1', \alpha_2'), (\alpha_1'', \alpha_2''))$ such that either $\alpha_1' = \alpha_1''$, or $\alpha_2' = \alpha_2''$, or α_2' is an ancestor of α_2''. An independent set of size d_1 in this graph consists of pairs (α_1', α_2') such that the first components are all the d_1 children of α_1 and the second components form an antichain among the descendants of α_2 to which the first components can map. If such an independent set exists, then we set $T(\alpha_1, \alpha_2)$ to true.

Let $n = |\mathsf{Addr}(t_1)|$ and $m = |\mathsf{Addr}(t_2)|$. To calculate $T(\alpha_1, \alpha_2)$ for each of the nm pairs α_1, α_2, we need to find an independent set in a graph with dm nodes, which can be solved with brute force in time $O(m^d)$, and this dominates the complexity. The total time of the algorithm is $O(nm^{d+1})$. □

7 IMPLEMENTATION

We implemented PIFO trees as described in §2, along with the embedding algorithm covered in Theorem 6.1, in OCaml. Our implementation includes a simulator that can "run" a traffic sample of packets through a PIFO tree and visualize the results with respect to push and pop time.

We show that our scheduler gives expected results when programming standard scheduling algorithms. We compare the traces from our scheduler to those obtained from a state-of-the-art hardware switch and find reasonable correlation. This reinforces the idea that PIFO trees are a reasonable primitive for standard programmable scheduling. We then move beyond the capabilities of a modern switch by showing the result of running hierarchical algorithms on our scheduler. Finally, we show how our implementation supports the embedding of one PIFO tree into another without having to program the latter tree anew and without appreciable loss in performance.

7.1 Preliminaries

A PIFO tree is programmed, as in the presentation thus far, by specifying a topology and a control. The simulator takes a collection of incoming packets, represented as a PCAP, and attempts to push the packets into the tree. It paces these pushes to match the packets' timestamps in the given PCAP. The simulator also requires a *line rate*, which is the frequency at which it automatically calls pop.

For ease of presentation, we have standardized a few things that are not fundamental to our implementation. All our traffic samples contain 60 packets of the same size, coming from seven source addresses that we label A through G. We partition the traffic into flows simply based on these addresses, though more sophisticated partitioning is of course possible.

An important subtlety is *saturation*: we need to enqueue packets quickly enough that the switch actually has some backlog of packets and therefore has to make a scheduling decision among those packets. This is easy to do in our OCaml simulator, where we have total control: we set packets to

arrive at the rate of 10 packets/s and simulate a slower line rate of 4 packets/s. We also achieve similar saturation when running comparative experiments on the hardware switch.

Reading our visualizations. In the visualizations that follow, the x-axis, read left to right, shows the time (in seconds) since the simulation started; the y-axis, read top to bottom, shows packets in the order they arrive at the scheduler. A colored horizontal bar represents a packet, and is colored based on its source address: A B C D E F G. A horizontal bar is drawn starting at the time when the packet arrives at the scheduler and ending at the time when it is released by the scheduler, so shorter bars indicate preferential service. Later packets tend to have longer bars precisely because we are saturating the scheduler. To notice trends generally, it is useful to zoom out and look at the "shadow" that a certain-colored flow casts. To focus more specifically on which packets are released when, it is useful to zoom in and study the right edges of the horizontal bars.

7.2 Standard Algorithms

We first show that PIFO trees are a reasonable primitive for standard, non-hierarchical algorithms. We use a simple ternary topology: one root node with three leaves below it. The standard algorithms discussed below correspond to the rows of Table 1. We have constructed a PCAP of packets with sources A B C, and, in the left column, we schedule that PCAP using PIFO trees.

The first come, first served (FCFS) algorithm transmits packets in the order they are received. The strict algorithm strictly prefers C to B and B to A, up to availability of input packets. It is easy to see the expected trend in the visualization: focus on the "shadow" casy by each flow; A's is the smallest, followed by B's, followed by C's. Round-robin seeks to alternate between flows A, B, and C. Weighted fair queueing (WFQ) [Demers et al. 1989] allows user-defined weights for each flow; in the table we show a split of 10/20/30 between A/B/C. The shadows guide intuition here as well.

As a further, albeit informal, endorsement of PIFO trees, we compare our tree-based scheduling trends to those of a modern switch. The right column of Table 1 shows the same algorithms and PCAPS scheduled by the FIFO-based programmable scheduler exposed by our hardware switch. We find the trends to be in agreement. Our testbed consists of two host Linux machines connected by a hardware switch. To keep the total number of packets small enough to present here, we use a slow ingress rate of 60 kbps and an egress rate of 24 kbps, with packets of size 1 kb. Although it is untenable to visualize them here, we also conduct similar experiments with more realistic ingress and egress rates (2.5 Mbps and 1 Mbps respectively) and observe similar scheduling trends.

We point out a few oddities in the hardware experiments:

(1) The first few packets going into the switch spend so little time in the queue that the horizontal bars representing them do not render in our visualizations.
(2) The switch releases packets not one by one but in batches of four. This trend does not appear in our more realistic experiments. We conjecture that this stagger is an artifact of our unrealistically slow rates: the egress thread appears to pull packets in batches.
(3) The packets do not arrive in as regular a pattern as in software; see the left edges of the horizontal bars. This is perhaps to be expected in a real-world experiment.

7.3 Hierarchical Algorithms

Our implementation supports arbitrary hierarchical PIFO trees; there is no equivalent in our programmable hardware switch. Hierarchical packet fair queueing (HPFQ) [Bennett and Zhang 1996] is WFQ applied at many levels, and is a more general instance of the motivating problem we discussed in §2.2. Table 2 shows HPFQ scheduling a traffic sample, now with unequal splits as shown. Paying attention to the right hand side of the horizontal colored bars, we see that B gets excellent service (80/20, versus C) until the first packets from A arrive, after which B's share decreases. This

Table 1. Standard scheduling algorithms as run against our scheduler and a programmable hardware switch.

	Through a PIFO tree	Through a programmable switch
FCFS		
Strict		
R'Robin		
WFQ		

Table 2. HPFQ running a traffic flow. Share ratios as indicated.

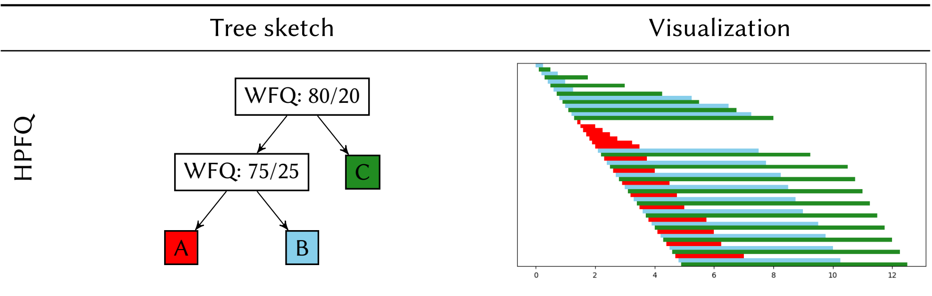
	Tree sketch	Visualization
HPFQ	WFQ: 80/20 → WFQ: 75/25, C; WFQ: 75/25 → A, B	

Table 3. Hierarchical algorithms (rows 1 and 3), and compiling ternary trees to binary (rows 2 and 4).

	Tree sketch	Visualization
TwoPol	WFQ: 10/10/80; A; B; Strict: E>D>C; C; D; E	0 2 4 6 8 10 12
TwoPol Bin	WFQ: 10/10/80; T; Strict: E>D>C; A; B; T; E; C; D	0 2 4 6 8 10 12
3Tier3	WFQ: 40/40/20; A; B; RR; C; D; WFQ: 10/40/50; E; F; G	0 2 4 6 8 10 12
3Tier3 Bin	WFQ: 40/40/20; T; RR; A; B; T; WFQ: 10/40/50; C; D; T; G; E; F	0 2 4 6 8 10 12

could not have been recreated using fair scheduling without hierarchies. Additionally, trees need not run the same algorithm on different nodes. The first and third rows of Table 3 show examples of more complicated trees running combinations of algorithms on their nodes.

7.4 Compilation

We implement a compilation from ternary PIFO trees to binary PIFO trees using the technique described in Theorem 6.1, meaning that we can take any ternary topology along with a control written against it, and automatically create a binary topology and control that together simulate

the former. We use this to compile all the ternary algorithms described in Table 1 into binary PIFO trees, and we find the resultant visualizations unchanged. We also compile the complex trees shown in rows 1 and 3 of Table 3 into binary trees; these are shown in rows 2 and 4. The transit nodes that were automatically created are shown in gray, and the visualizations produced remain unchanged.

Our visualizations are sensitive to push and pop time, and so it is encouraging to see them stay the same across the compilation process: this suggests that we experience no appreciable loss in performance as a result of the compilation despite the introduction of new intermediary nodes.

8 RELATED WORK

We outline a class of algorithms that we have not considered in our formalization. We review other efforts in formalizing schedulers. We study other efforts in programmable packet scheduling.

8.1 Non Work-Conserving Algorithms

The focus of our study has been *work-conserving* scheduling: a pushed packet can immediately be popped. *Non work-conserving* algorithms, also known as shaping algorithms, say that a packet cannot be popped until some *time* that is computed, specifically for that packet, at push. This means that a less favorably ranked packet may be released before a more favorably ranked one if the former is ready and the latter is not, and the link may go idle if no packets are ready. We leave shaping for future work, but provide a few pointers. Sivaraman et al. [2016b] include shaping in their PIFO tree model. Loom [Stephens et al. 2019] repeatedly reinserts shaped packets into the tree. Carousel [Saeed et al. 2017] takes a more general approach, applying shaping to all algorithms.

8.2 Formalizations in the Domain of Packet Scheduling

There is a wealth of work from the algorithms and theory communities towards formally studying packet scheduling using competitive analysis [Aiello et al. 2005; Kesselman et al. 2004; Mansour et al. 2004]. We refer interested readers to a comprehensive survey by Goldwasser [2010].

Chakareski [2011] formalizes in-network packet scheduling for video transmission as a constrained optimization problem expressed using Lagrange multipliers. Nodes coordinate to compute the optimal rate at which other nodes should send packets. Dürr and Nayak [2016] map packet scheduling to the no-wait job-shop scheduling problem from operations research, arriving at an integer linear program that exposes the constraints under which to minimize maximum congestion.

SP-PIFOs [Alcoz et al. 2020] orchestrate a collection of FIFOs to approximate the behavior of a PIFO. This is bound to be imperfect, so they give a formal way to measure the number of mistakes their model makes relative to a perfect PIFO. Follow-on analysis by Vass et al. [2022] problematizes Alcoz et al.'s push-up/push-down heuristic, showing that it can introduce mistakes linearly up to the number of FIFOs. Their solution is a new heuristic called Spring that counts packets using exponentially weighted moving averages and can achieve twofold speedup compared to SP-PIFOs.

In search of a universal packet scheduler, Mittal et al. [2016] develop a general model of packet scheduling. They formally define a *schedule* as a set of the (fixed set of) packets, the paths those packets take through the network, and the input and output times the packets enjoy. A schedule is effected by the set of scheduling algorithms that the routers along the way implement. One set of scheduling algorithms can *replay* another if, for all packets in the fixed set, it gives each packet an equal or better output time. Mittal et al. show that there is no universal set of algorithms that can replay all others, but that the classic least slack time first algorithm [Leung 1989] comes close.

8.3 Programmable Scheduling

We outline other lines of work that also orchestrate FIFOs and PIFOs to eke out more expressivity. We also review work that allows higher-level programming of schedulers.

Orchestrating FIFOs and PIFOs. SP-PIFOs [Alcoz et al. 2020], orchestrate a collection of FIFOs to approximate the behavior of a PIFO. Sharma et al. [2018] approximate WFQ on reconfigurable switches. They present a scheduler called rotating strict priority, which transmits packets from multiple queues in approximate sorted order by grouping FIFOs into two groups and intermittently switching the relative priority of one group versus the other. Tonic [Arashloo et al. 2020] can effect a range of scheduling algorithms using a FIFO along with the added notion of *credit* to implement the round-robin algorithm. Zhang et al. [2021] study packet scheduling using the push-in pick-out (PIPO) data structure. A PIPO is an approximation of the push-in extract-out (PIEO) queue [Shrivastav 2019] which picks out the best-ranked item that also satisfies some pop-time predicate. Sivaraman et al. [2016b] orchestrate PIFOs into a tree structure. They provide a hardware design targeting shared memory switches. They explain how a PIFO tree can be laid out in memory as a complete mesh, and sketch a compiler from the logical tree to the theoretical hardware mesh.

Programming at a higher level. QtKAT [Schlesinger et al. 2015] extends NetKAT [Anderson et al. 2014] to bring quality of service into consideration. Their work paves the way for formal analysis using network calculus and, eventually, the verification of network-wide queueing. Arashloo et al. [2016] present SNAP, a tool that allows a "one big switch" abstraction: users can reason at a global level and program a fictitious big switch, and the tool checks the program for correctness and compiles it to the distributed setting. NUMFabric [Nagaraj et al. 2016] allows the user to specify a utility maximization problem, once, and it then calculates the allocations of bandwidth that would maximize the utility function across the distributed network. Sivaraman et al. [2016a] allow data-plane programming as a high level "packet transaction" written in Domino, a new imperative language that compiles to line-rate hardware-level code running on programmable switches.

9 CONCLUSION AND FUTURE WORK

This paper explored higher-level abstractions for packet scheduling. Starting with a proposal by Sivaraman et al. [2016b], we formalized the syntax and semantics of PIFO trees, developed alternate characterizations in terms of permutations on packets, established expressiveness results. We also designed, implemented and tested embedding algorithms. Overall, we believe our work represents the next step toward developing a programming language account of scheduling algorithms—an important topic that has mostly remained in the domain of networking and systems. In the future, we are interested in further exploring the theory and practice of scheduling algorithms, including non-work conserving algorithms as well as applications to other domains, such as task scheduling.

10 DATA AVAILABILITY STATEMENT

Code supporting this paper is maintained publicly on GitHub [Mohan et al. 2023a]. The version submitted to the OOPSLA '23 AEC is permanently archived on Zenodo [Mohan et al. 2023c]. A version of this paper that includes proofs of lemmas in §4 and §5 is on arXiv [Mohan et al. 2023b].

ACKNOWLEDGMENTS

Thanks to Éva Tardos for suggesting the more general embedding algorithm for arbitrary target trees. This work was supported by the NSF under award 2118709, grant CCF-2008083, and grant FMiTF-1918396, the ONR under contract N68335-22-C-0411, and DARPA under contract HR001120C0107. T. Kappé was partially supported by the EU's Horizon 2020 research and innovation program under Marie Skłodowska-Curie grant VERLAN (101027412).

REFERENCES

William Aiello, Yishay Mansour, S. Rajagopolan, and Adi Rosén. 2005. Competitive queue policies for differentiated services. *J. Algorithms* 55, 2 (2005), 113–141. https://doi.org/10.1016/j.jalgor.2004.04.004

Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. 2020. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *NSDI*. USENIX Association, Berkeley, California, 59–76. https://www.usenix.org/conference/nsdi20/presentation/alcoz

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: semantic foundations for networks. In *POPL*. ACM, New York, 113–126. https://doi.org/10.1145/2535838.2535862

Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *SIGCOMM*. ACM, New York, 29–43. https://doi.org/10.1145/2934872.2934892

Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *NSDI*. USENIX Association, Berkeley, California, 93–109. https://www.usenix.org/conference/nsdi20/presentation/arashloo

Jon C. R. Bennett and Hui Zhang. 1996. Hierarchical Packet Fair Queueing Algorithms. In *SIGCOMM*. ACM, New York, 143–156. https://doi.org/10.1145/248156.248170

Jacob Chakareski. 2011. In-Network Packet Scheduling and Rate Allocation: A Content Delivery Perspective. *IEEE Trans. Multim.* 13, 5 (2011), 1092–1102. https://doi.org/10.1109/TMM.2011.2157673

Thomas M. Cover and Joy A. Thomas. 2006. *Elements of information theory*. Wiley, Hoboken, New Jersey.

Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: consensus at network speed. In *SOSR*. ACM, New York, 5:1–5:7. https://doi.org/10.1145/2774993.2774999

Keri D'Angelo and Dexter Kozen. 2023. Abstract Huffman Coding and PIFO Tree Embeddings. In *Data Compression Conference, DCC 2023*. IEEE, 1. https://doi.org/10.1109/DCC55655.2023.00077

Alan J. Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM*. ACM, New York, 1–12. https://doi.org/10.1145/75246.75248

Frank Dürr and Naresh Ganesh Nayak. 2016. No-wait Packet Scheduling for IEEE Time-sensitive Networks (TSN). In *RTNS*. ACM, New York, 203–212. https://doi.org/10.1145/2997465.2997494

Nate Foster, Nick McKeown, Jennifer Rexford, Guru M. Parulkar, Larry L. Peterson, and M. Oguz Sunay. 2020. Using deep programmability to put network owners in control. *Comput. Commun. Rev.* 50, 4 (2020), 82–88. https://doi.org/10.1145/3431832.3431842

Michael H. Goldwasser. 2010. A survey of buffer management policies for packet switches. *SIGACT News* 41, 1 (2010), 100–128. https://doi.org/10.1145/1753171.1753195

Intel. 2022. *Product Brief: Intel Tofino 3 Intelligent Fabric Processors*. Technical Report. Intel. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-05/tofino-3-intelligent-fabric-processors-brief.pdf

Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *NSDI*. USENIX Association, Berkeley, California, 35–49. https://www.usenix.org/conference/nsdi18/presentation/jin

Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP*. ACM, New York, 121–136. https://doi.org/10.1145/3132747.3132764

Alexander Kesselman, Zvi Lotker, Yishay Mansour, Boaz Patt-Shamir, Baruch Schieber, and Maxim Sviridenko. 2004. Buffer Overflow Management in QoS Switches. *SIAM J. Comput.* 33, 3 (2004), 563–583. https://doi.org/10.1137/S0097539701399666

Jon M. Kleinberg and Éva Tardos. 2006. *Algorithm design*. Addison-Wesley, Boston, Massachusetts.

Joseph Y.-T. Leung. 1989. A New Algorithm for Scheduling Periodic Real-Time Tasks. *Algorithmica* 4, 2 (1989), 209–219. https://doi.org/10.1007/BF01553887

Bingzhe Liu, Ali Kheradmand, Matthew Caesar, and Philip Brighten Godfrey. 2020. Towards Verified Self-Driving Infrastructure. In *HotNets*. ACM, New York, 96–102. https://doi.org/10.1145/3422604.3425949

Yishay Mansour, Boaz Patt-Shamir, and Ofer Lapid. 2004. Optimal smoothing schedules for real-time streams. *Distributed Comput.* 17, 1 (2004), 77–89. https://doi.org/10.1007/s00446-003-0101-0

Robin Milner. 1971. An Algebraic Definition of Simulation Between Programs. In *IJCAI*. ACM, New York, 481–489. http://ijcai.org/Proceedings/71/Papers/044.pdf

Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Universal Packet Scheduling. In *NSDI*. USENIX Association, Berkeley, California, 501–521. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/mittal

Anshuman Mohan, Yunhe Liu, Nate Foster, Tobias Kappé, and Dexter Kozen. 2023a. Code repository in support of 'Formal Abstractions for Packet Scheduling'. https://github.com/cornell-netlab/pifo-trees-artifact

Anshuman Mohan, Yunhe Liu, Nate Foster, Tobias Kappé, and Dexter Kozen. 2023b. Formal Abstractions for Packet Scheduling: Extended Version. (2023). https://arxiv.org/abs/2211.11659

Anshuman Mohan, Yunhe Liu, Nate Foster, Tobias Kappé, and Dexter Kozen. 2023c. Software artifact in Support of 'Formal Abstractions for Packet Scheduling'. https://doi.org/10.5281/zenodo.8329703

Kanthi Nagaraj, Dinesh Bharadia, Hongzi Mao, Sandeep Chinchali, Mohammad Alizadeh, and Sachin Katti. 2016. NUMFabric: Fast and Flexible Bandwidth Allocation in Datacenters. In *SIGCOMM*. ACM, New York, 188–201. https://doi.org/10.1145/2934872.2934890

Jan J. M. M. Rutten. 2000. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.* 249, 1 (2000), 3–80. https://doi.org/10.1016/S0304-3975(00)00056-6

Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable Traffic Shaping at End Hosts. In *SIGCOMM*. ACM, New York, 404–417. https://doi.org/10.1145/3098822.3098852

Cole Schlesinger, Hitesh Ballani, Thomas Karagiannis, and Dimitrios Vytiniotis. 2015. *Quality of Service Abstractions for Software-defined Networks*. Technical Report. Microsoft Research. https://www.microsoft.com/en-us/research/publication/quality-of-service-abstractions-for-software-defined-networks/

Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *NSDI*. USENIX Association, Berkeley, California, 1–16. https://www.usenix.org/conference/nsdi18/presentation/sharma

Vishal Shrivastav. 2019. Fast, scalable, and programmable packet scheduler in hardware. In *SIGCOMM*. ACM, New York, 367–379. https://doi.org/10.1145/3341302.3342090

Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016a. Packet Transactions: High-Level Programming for Line-Rate Switches. In *SIGCOMM*. ACM, New York, 15–28. https://doi.org/10.1145/2934872.2934900

Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016b. Programmable Packet Scheduling at Line Rate. In *SIGCOMM*. ACM, New York, 44–57. https://doi.org/10.1145/2934872.2934899

Brent E. Stephens, Aditya Akella, and Michael M. Swift. 2019. Loom: Flexible and Efficient NIC Packet Scheduling. In *NSDI*. USENIX Association, Berkeley, California, 33–46. https://www.usenix.org/conference/nsdi19/presentation/stephens

Balázs Vass, Csaba Sarkadi, and Gábor Rétvári. 2022. Programmable Packet Scheduling With SP-PIFO: Theory, Algorithms and Evaluation. In *INFOCOM Workshops*. IEEE, New York, 1–6. https://doi.org/10.1109/INFOCOMWKSHPS54753.2022.9798055

Chuwen Zhang, Zhikang Chen, Haoyu Song, Ruyi Yao, Yang Xu, Yi Wang, Ji Miao, and Bin Liu. 2021. PIPO: Efficient Programmable Scheduling for Time Sensitive Networking. In *ICNP*. IEEE, New York, 1–11. https://doi.org/10.1109/ICNP52444.2021.9651944

Received 2023-04-14; accepted 2023-08-27