Multi-Queue Fair Queuing

Mohammad Hedayati, ***University of Rochester;*** **Kai Shen,** ***Google;***
Michael L. Scott, ***University of Rochester;*** **Mike Marty,** ***Google***

https://www.usenix.org/conference/atc19/presentation/hedayati-queue

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

Multi-Queue Fair Queueing

Mohammad Hedayati
University of Rochester

Kai Shen
Google

Michael L. Scott
University of Rochester

Mike Marty
Google

Abstract

Modern high-speed devices (e.g., network adapters, storage, accelerators) use new host interfaces, which expose multiple software queues directly to the device. These *multi-queue* interfaces allow mutually distrusting applications to access the device without any cross-core interaction, enabling throughput in the order of millions of IOP/s on multicore systems. Unfortunately, while independent device access is scalable, it also introduces a new problem: unfairness. Mechanisms that were used to provide fairness for older devices are no longer tenable in the wake of multi-queue design, and straightforward attempts to re-introduce it would require cross-core synchronization that undermines the scalability for which multiple queues were designed.

To address these challenges, we present Multi-Queue Fair Queueing (MQFQ), the first fair, work-conserving scheduler suitable for multi-queue systems. Specifically, we (1) reformulate a classical fair queueing algorithm to accommodate multi-queue designs, and (2) describe a scalable implementation that bounds potential unfairness while minimizing synchronization overhead. Our implementation of MQFQ in Linux 4.15 demonstrates both fairness and high throughput. Evaluation with an NVMe over RDMA fabric (NVMf) device shows that MQFQ can reach up to 3.1 Million IOP/s on a single machine—20× higher than the state-of-the-art Linux Budget Fair Queueing. Compared to a system with no fairness, MQFQ reduces the slowdown caused by an antagonist from 3.78× to 1.33× for the FlashX workload and from 6.57× to 1.03× for the Aerospike workload (2× is considered "fair" slowdown).

1 Introduction

Recent years have seen the proliferation of very fast devices for I/O, networking, and computing acceleration. Commodity solid-state disks (e.g., Intel Optane DC P4800X [22] or Samsung PM1725a [38]) can perform at or near a million I/O operations per second. System-area networks (e.g., InfiniBand) can sustain several million remote operations per second over a single link [25]. RDMA delivers data across fabric within a few microseconds. GPUs and machine learning accelerators may offload computations that run just a few microseconds at a time [30]. At the same time, the proliferation of multicore processors has necessitated architectures tuned for independent I/O across multiple hardware threads [4, 36].

These technological changes have shifted performance bottlenecks from hardware resources to the software stacks that manage them. In response, it is now common to adopt a *multi-queue* architecture in which each hardware thread owns a dedicated I/O queue, directly exposed to the device, giving it an independent path over which to send and receive requests. Examples of this architecture include multi-queue SSDs [22, 38, 50] and NICs [42], and software like the Windows and Linux NVMe drivers, the Linux multi-queue block layer [5], SCSI multi-queue support [8], and data-plane OSes [4, 36]. A recent study [51] demonstrated up to 8× performance improvement for YCSB-on-Cassandra, using multi-queue NVMe instead of single-queue SATA.

To support the full bandwidth of modern devices, multi-queue I/O systems are designed to incur no cache-coherence traffic in the common case when sending and receiving requests. It's easy to see why: a device supporting millions of IOP/s sees each new request in a fraction of a microsecond—a time interval that allows for fewer than 10 cross-core cache coherence misses, and is comparable to the latency of a single inter-processor interrupt (IPI). Serializing requests at such high speeds is infeasible now, and will only become more so as device speeds continue to increase while single-core performance stays relatively flat. As a result, designers have concluded that conventional fair-share I/O schedulers, including fair queueing approaches [35, 40], which reorder requests in a single queue, are unsuited for modern fast devices.

Unfortunately, by cutting out the OS resource scheduler, direct multi-queue device access undermines the OS's traditional responsibility for fairness and performance isolation. While I/O devices (e.g., SSD firmware, NICs) may multiplex hardware queues, their support for fairness is hampered by their inability to reason in terms of system-level policies for resource principals (applications, virtual machines, or Linux

cgroups), or to manage an arbitrary number of flows. As a result, device-level scheduling tends to cycle naively among I/O queues in a round robin fashion [44]. Given such simple scheduling, a greedy application or virtual machine may gain unfair advantage by issuing I/O operations from many CPUs (so it can obtain resource shares from many queues). It may also gain advantage by "batching" its work into larger requests (so more of its work gets done in each round-robin turn). Even worse, a malicious application may launch a denial-of-service attack by submitting a large number of artificially created expensive requests (e.g., very large SSD writes) through many or all command queues.

As a separate issue, it is common for modern SSDs [9, 44] and accelerators [20, 32] to support parallel requests internally. Traditional resource scheduling algorithms, which assume underlying serial operation, are unsuitable for devices with a high degree of internal parallelism.

To overcome these problems, we present *Multi-Queue Fair Queueing (MQFQ)*—the first fair scheduler, to the best of our knowledge, capable of accommodating multi-queue devices with internal parallelism in a scalable fashion. As in classical *fair queueing* [13, 34], we ensure that each *flow* (e.g., an application, virtual machine, or Linux cgroup) receives its share of bandwidth. While classical fair queueing employs a single serializing request queue, we adapt the fair queueing principle to *multi-queue* systems, by efficiently tracking global resource utilization and arranging to *throttle* any queue that has exceeded its share by some bounded amount.

Accordingly, we introduce a *throttling threshold* T such that each core can dispatch, without coordinating with other cores, as long as the lead request in its queue is within T of the utilization of the slowest active queue, system-wide. This threshold creates a window within which request dispatches can *commute* [10], enabling scalable dispatch. We show mathematically that this relaxation has a bounded impact on fairness. When $T = 0$, the guarantees match those of classical fair queueing.

The principal obstacle to scalability in MQFQ is the need for cross-queue synchronization to track global resource utilization.We demonstrate that it is possible, by choosing appropriate data structures, to sustain millions of IOP/s while guaranteeing fairness. The key to our design is to *localize* synchronization (intra-core rather than inter-core; intra-socket rather than inter-socket) as much as possible. An instance of the *mindicator* of Liu et al. [29] allows us to track flows' shares without a global cache miss on every I/O request. A novel data structure we call the *token tree* allows us to track available internal device parallelism: an I/O completion frees up a slot that is preferentially reused by the local queue if possible; otherwise, our token tree allows fast reallocation to a nearby queue. Finally, a nonblocking variant of a timer wheel [43, 47] keeps track of queues whose head requests are too far ahead of the shares of their contributing flows: when resource utilization has advanced sufficiently, update of a single index suffices to turn the wheel and unblock the appropriate flows. MQFQ demonstrates that while scalable multi-queue I/O precludes serialization, it can tolerate infrequent, physically localized synchronization, allowing us to achieve both fairness and high performance.

Summarizing contributions:

- We present Multi-Queue Fair Queueing—to the best of our knowledge, the first scalable, fair scheduler for multi-queue devices.
- We demonstrate mathematically that adapting the fair queueing principle to multi-queue devices results in a bounded impact on fairness.
- We introduce the *token tree*, a novel data structure that tracks available dispatch slots in a multi-queue device with internal parallelism.
- We present a scalable implementation of MQFQ. Our implementation uses the token tree along with two other scalable data structures to localize synchronization as much as possible.

2 Background and Design

Fair queueing [13, 34] is a class of algorithms to schedule a network, processing, or I/O resource among competing flows. Each flow comprises a sequence of requests or packets arriving at the device. Each request has an associated cost, which reflects its resource usage (e.g., service time or bandwidth). Fair queueing then allocates resources in proportion to weights assigned to the competing flows.

A flow is said to be *active* if it has any requests in the system (either waiting to be dispatched to the device, or waiting to be completed *in* the device), and *backlogged* if it is active and has at least one outstanding request to be dispatched. Fair queueing algorithms are *work-conserving*: they schedule requests to consume surplus resources in proportion to the weights of the active flows. A flow whose requests arrive too slowly may become inactive and forfeit the unconsumed portion of its share.

Start-time Fair Queueing (SFQ) [18, 19] assigns a start and finish tag to each request when it arrives, and dispatches requests in increasing order of start tags; ties are broken arbitrarily. The tag values represent the point in the history of resource usage at which each request should start and complete according to a system notion of *virtual "time."* Virtual time always advances monotonically and is identical to real time if: (1) all flows are backlogged, (2) the device (server) completes work at a fixed ideal rate, (3) request costs are an accurate measure of service time, and (4) the weights sum to the service capacity. The start tag for a request is set to be the maximum of the virtual time at arrival and the last finish tag of the flow. The finish tag for a request is its start tag plus its cost, normalized to the weight of the flow.

When the server is busy, virtual time is defined to be equal to the start tag of the request in service, and when it is idle, maximum finish tag of any request that has been serviced by

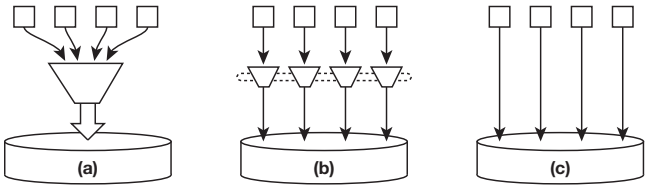

Figure 1: MQFQ (b) employs a set of per-CPU priority queues, rather than (a) a single central queue or (c) fully independent access. Queues coordinate through scalable data structures (suggested by the dotted line; described in Sec. 3) to maintain fairness.

that time. Note that this definition assumes at most a single request can be in service at any moment.

Parallel Dispatch A server with internal parallelism may service multiple requests simultaneously, so virtual time as defined in SFQ is not well-defined in this setting. Moreover, even an active flow may lag behind in resource utilization if it generates an insufficient number of concurrent requests to consume its assigned share.

SFQ(D) [23] works the same as SFQ but allows up to D in-service requests ($D = 1$ reduces to SFQ). Due to out-of-order completion, for the busy server case, virtual time is redefined to be the start tag of the last dispatched request. Note that this definition requires requests to be dispatched in increasing order of start tags, which precludes scalable implementation on multi-queue systems.

2.1 Multi-Queue Fair Queueing

The main obstacle in adapting fair queueing—or most other scheduling algorithms, for that matter—to a multi-queue I/O architecture is the need to dispatch requests in an order enforced by a central priority queue. Additional challenges include the need to dispatch multiple requests concurrently (to saturate an internally parallel device) and the inability to simply advance virtual "time" on dispatch or completion events, since these may occur out of order.

In MQFQ, we replace the traditional central priority queue (Fig. 1(a)) with a set of per-CPU priority queues (Fig. 1(b)), each of which serves to order local requests. To limit imbalance across queues, we suspend (*throttle*) any queue whose lead request is ahead of the slowest backlogged flow in the system (the one that determines the virtual time) by more than some predefined threshold T, allowing other queues to catch up. Setting $T = 0$, while limits scalability in practice, would effectively restore the semantics of a global priority queue. Setting $T > 0$ leads to relaxed semantics but lower synchronization overhead by utilizing the *Scalable Commutativity Rule* [10] to allow requests dispatches to be reordered, i.e., to commute. Specifically, it allows for windows of *conflict free* operations (i.e., no core writes a cache line that was read or written by another core) enabling scalable implementation. While short-term fluctuations of as much as T in the relative progress of flows is possible, it still preserves long-term shares. By adjusting T appropriately, we can find a design point that provides most of the fairness of traditional fair queueing with most of the performance of fully independent queues.

For an internally parallel device, in order to keep the device busy, we will often need to dispatch a new request before the previous one has finished. At the same time, since the device decides the order in which dispatched requests are served, we must generally avoid dispatching more requests than can actually be handled in parallel, thereby preserving our ability to order them. We therefore introduce a second parameter, D, which represents the maximum number of outstanding dispatched requests across all queues.

Recall that a *backlogged* flow is one that has requests ready to be dispatched, and an *active* flow is one that is either backlogged or has requests pending in the device. For any device that supports $D \geq 2$ concurrent requests, the distinction between backlogged and active is quite important: it is no longer the case that an active flow is using at least its fair share (i.e., the flow could be *non-saturating*). In a traditional fair queueing system, an active flow determines the progression of virtual time. With a parallel device, this convention would allow a non-saturating active flow to keep idle resources from being allotted to other flows, leading to underutilization. To fix this, a scheduler aware of internal parallelism needs to use backlogged (instead of active) flows to determine virtual time. We therefore define virtual time (and thus the start tag of a newly arriving request on a previously non-backlogged flow) to be the minimum start tag of all requests across all queues. In a multi-queue system, computing this global minimum without frequent cache misses is challenging. In Sec. 3.1 we show how we localize the misses using a mindicator [29].

The lack of a central priority queue, and our use of the throttling threshold T, raises the possibility not only that requests will complete out of order, but that they may be *dispatched* out of order.

We now define our notion of per-flow virtual time, in a way that accommodates the internal parallelism of the device while retaining the essential property that a *lagging* flow (i.e., a flow that is not backlogged) can never accumulate resources for future use. Recall that queues hold requests that have been submitted but not yet dispatched to the device. The flows that submitted these requests are backlogged by definition. For each such flow f, its virtual time is defined to be the start tag of f's first (oldest) backlogged (waiting to be dispatched) request. (Note that f may have backlogged requests in multiple queues.) Assuming f has multiple pending requests, dispatching this first request would increase f's virtual time by l/r, where r is f's weight (its allotted share of the device) and l is the length (size) of the request. (For certain devices we may also scale the "size" in accordance with operation type—e.g., to reflect the fact that writes are more expensive than reads on an SSD.)

We define global virtual time to be the minimum of per-flow virtual times across all backlogged flows. This is the same

as the minimum of the start tags of the lead requests across all queues, since requests in each queue are sorted by start tags. This equivalence allows us to ignore the maintenance of per-flow virtual times; instead, we directly maintain the global virtual time (hereafter, simply "virtual time") as the minimum start tag of the lead requests across all queues.

As soon as a flow becomes lagging, it stops contributing to the virtual time, which may advance irrespective of a lack of activity in the lagging flow. Request start tags from a lagging flow are still subject to the lower bound of current virtual time. MQFQ then ensures that no request is dispatched if its start tag exceeds the virtual time by more than T. To throttle a flow f that has advanced too far, it suffices to throttle any queues headed by f's requests: since requests in each queue are sorted by start tags, all other requests in such a queue are also guaranteed to be more than T ahead of virtual time.

High-level pseudocode for MQFQ appears in Fig. 2.

```
global structures:
   VT mindicator
   wheel of throttled queues
   token tree of available slots
   set of ready queues (nonempty, unthrottled)

per-flow structures:
   end tag of last submitted request

per-CPU structures:
    local queue of not-yet-dispatched requests

on submission of request R:
   set R's start tag = MAX(VT, per-flow end tag)
   set R's end tag =
      R's start tag + R's service time
   update per-flow end tag
   insert R in local queue
   if R goes at the head
      update VT
   dispatch()

dispatch():
   if local queue is in throttling wheel
      remove it from wheel
   if local queue is in ready queues
      remove it from ready queues
   if local queue is empty
      return
   for lead request R from local queue
      if R's start tag is more than T ahead of VT
         add local queue to throttling wheel
         return
      attempt to obtain slot from token tree
      if unsuccessful
         add local queue to set of ready queues
         return
      remove R from local queue
      deliver R to device
      update VT
      if VT has advanced a bucket's worth
         turn the throttling wheel
         unblock any no-longer-throttled queues
            for which slots are readily available
         add the rest to the set of ready queues

on unblock:
   dispatch()

on request completion:
   choose nearest Q in ready queues (could be self)
   return slot to token tree w.r.t. Q
   unblock Q
```

Figure 2: High-level pseudocode for the MQFQ algorithm. Logic to mitigate races has been elided, as have certain optimizations (e.g., to avoid pairs of data structure changes that cancel one another out).

2.2 Fairness Analysis

If flows have equal weight, allocation of the device is fair if equal bandwidth is allocated to each (backlogged) flow in every time interval. With unequal weights, each backlogged flow should receive bandwidth proportional to its weight.

If we represent the weight of flow f as r_f and the service (in bytes) that it receives in the interval $[t_1,t_2]$ as $W_f(t_1,t_2)$, then an allocation is fair if for every time interval $[t_1,t_2]$, for every two backlogged flows f and m, we have:

$$\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m} = 0$$

Clearly, this is possible only if the flows can be broken into infinitesimal units. For a packet- or block-based resource we want

$$\left|\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m}\right| \leq H(f,m)$$

to be as close to 0 as possible. $H(f,m)$ is a function of the maximum request lengths, l_f^{max} and l_m^{max}, of flows f and m. Golestani [17] derives a lower bound on the fairness of any scheduler with single dispatch:

$$H(f,m) \geq \frac{1}{2}\left(\frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m}\right)$$

We similarly derive bounds on the fairness achieved by MQFQ. Our analysis builds on the fairness bounds for Start-time Fair Queueing (SFQ) [18] and SFQ(D) [23]. Goyal et al. [18] have previously shown in SFQ that in any interval for which flows f and m are backlogged during the entire interval, the difference of weighted services received by two flows at an SFQ server, given as:

$$\left|\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m}\right| \leq \frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m}$$

is twice the lower bound. SFQ uses a single priority queue and serves one request at a time. Now consider an otherwise unchanged variant of SFQ in which the single priority queue is replaced by multiple priority queues with throttled dispatch. We service one request at a time, which can come from any of the queues so long as its start tag is less than or equal to the global minimum + T. We call this variant Multi-Queue Fair Queueing with single dispatch—MQFQ(1).

Theorem 1 *For any interval in which flows f and m are backlogged during the entire interval, the difference in weighted services received by two flows at an MQFQ(1) server with throttling threshold T is:*

$$\left|\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m}\right| \leq 2T + \frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m}$$

We sketch a proof of Theorem 1 as follows.

Lemma 1 (Lower bound of service received by a flow): *If flow f is backlogged throughout the interval $[t_1,t_2]$, then in an MQFQ(1) server with throttling threshold T:*

$$W_f(t_1,t_2) \geq r_f \cdot (v_2 - T - v_1) - l_f^{max}$$

where v_1 is virtual time at t_1 and v_2 is virtual time at t_2.

Lemma 1 is true since at t_2 any backlogged flow has dispatched all requests whose start tag $\leq v_2 - T$. Only the last request may be outstanding at t_2—i.e., all but the last request must have completed. Since the last request's size is at most l_f^{max}, the finish tag of the last completed request must be at least $v_2 - T - l_f^{max}/r_f$. Therefore if we just count the completed requests in $[t_1,t_2]$, the minimum service received by backlogged flow f is at least $r_f \cdot (v_2 - T - v_1) - l_f^{max}$.

Lemma 2 (Upper bound of received service by a flow): *If flow f is backlogged throughout the internal $[t_1,t_2]$, then in an MQFQ(1) system with throttling threshold T:*

$$W_f(t_1,t_2) \leq r_f \cdot (v_2 + T - v_1) + l_f^{max}$$

Lemma 2 is true since at t_2 flow f may have, at most, dispatched all requests with start tag $\leq v_2 + T$. In the maximum case, the last completed request's finish tag will be no more than $v_2 + T$. In addition, one more request of size at most l_f^{max} may be outstanding and, in the maximum case, almost entirely serviced. Counting the completed requests and the outstanding request, the maximum service received by flow f is at most $r_f \cdot (v_2 + T - v_1) + l_f^{max}$.

Unfairness is maximized when one flow receives its upper bound of service while another flow receives its lower bound. Therefore, unfairness in MQFQ(1) with throttling threshold T is bounded by

$$\left| \frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m} \right| \leq 2T + \frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m}$$

This completes the proof of Theorem 1. ∎

Note that when $T = 0$, MQFQ(1) provides the same fairness bound as SFQ. Therefore T represents a tradeoff between fairness and scalability in a multi-queue system.

If we allow $D > 1$ parallel dispatches in an MQFQ(D) server, the fairness bound changes as follows:

Theorem 2 *In any interval for which flows f and m are backlogged during the entire interval, the difference of weighted services received by the two flows at an MQFQ(D) server with throttling threshold T is given as:*

$$\left| \frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m} \right| \leq (D+1)\left(2T + \frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m}\right)$$

This is true based on a combination of Theorem 1 and the proved fairness bound for SFQ(D) [23]. We omit the detailed proof. When the throttling threshold $T = 0$, MQFQ(D) provides the same fairness bound as SFQ(D).

3 Scalability

MQFQ employs a separate priority queue for every CPU (hardware thread), to minimize coherence misses and maximize scalability. A certain amount of sharing and synchronization is required, however, to maintain fairness across queues. Specifically, we need to track (1) the progression of virtual time; (2) the number of available I/O slots and the queues that can use them; and (3) the state of queues (throttled or not) and when they should be unthrottled. Our guiding principle is to maximize locality wherever possible. So long as utilization and fairness goals are not violated, we prefer to dispatch from the local queue, queues on the same core, queues on the same socket, and queues on another socket, in that order.

3.1 Virtual Time

Virtual time in MQFQ reflects resource usage (e.g., bandwidth consumed), and not wall-clock time. When a flow transitions from lagging to backlogged, the request responsible for the transition is set to have its start tag equal to current virtual time. As long as the flow remains backlogged, its following requests get increasing start tags with respect to the flow's resource usage: the start tag of each new request is set to the end tag of the previous request. Virtual time, in turn, is the minimum start tag of any request across all queues.

Naively, one might imagine an array, indexed by queue, with each slot indicating the start tag of its queue's lead request (if any). We could then compute the global virtual time by scanning the array. Such a scan, however, is far too expensive to perform on a regular basis (see Sec. 4.3.1). Instead, we use an instance of Liu et al.'s *mindicator* structure [29], modified to preclude decreases in the global minimum. The mindicator is a tree-based structure reminiscent of a priority-queue heap. Each queue is assigned a leaf in the tree; each internal node indicates the minimum of its children's values. A flow whose virtual time changes updates its leaf and, if its previous value was the minimum among its siblings, propagates the update root-ward. Changes reach the root only when the global minimum changes. While this is not uncommon (time continues to advance, after all), many requests in a highly parallel device complete a little out of order, and the mindicator achieves a significant reduction in contention.

Within each flow, we must also track the largest finish tag across all threads. For this we currently employ a simple shared integer value, updated (with fetch-and-add) on each request dispatch. In future work, we plan to explore whether better performance might be obtained with a scalable monotonic counter [6, 14], at least for flows with many threads.

3.2 Available Slots

A queue in MQFQ is unable to dispatch either when it is too far ahead of virtual time or when the device is saturated. For the latter case, MQFQ must track the number of outstanding (dispatched but not yet completed) requests on the device. Ideally, we want to dispatch exactly as many requests as the

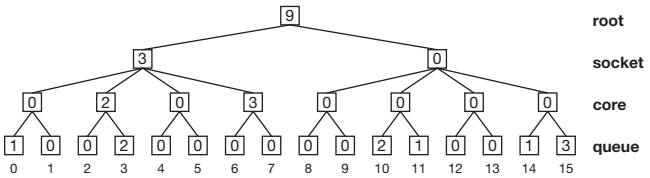

Figure 3: Example token tree for a 2-socket, 4-core-per, 2-thread-per machine. Values indicate currently unused device capacity. (If the device were fully subscribed [D outstanding requests], all values in the tree would be zero.) In the figure, there are 3 slots immediately available to queue 15. Queues 6 or 7 could use capacity allocated to their core; queues 4 or 5 could use capacity allocated to their socket; queues 8 or 9 would need to use capacity from the root.

device can handle in parallel, thereby avoiding any buildup in the device and preserving our ability to choose the ideal request to submit when an outstanding request completes.

We find (see Sec. 4.3.3) that a naive single shared cache line, atomically incremented and decremented upon dispatch and completion of requests, fails to scale when many queues are frequently trying to update its value. We therefore aim to improve locality by preferentially allocating available slots to physically nearby queues, in a manner reminiscent of cohort locks [15]. This approach meshes well with our notification mechanism, which prefers to unblock nearby queues.

As a compromise between locality and flexibility, we have implemented a structure we call the *token tree* (Fig. 3). Values in leaves represent unused capacity ("slots") currently allocated to a given local queue. Parent nodes represent additional capacity allocated to a given core, and so on up the tree. The values of all nodes together sum to the difference between D and the number of active requests on the device. When we need to dispatch a request, we try to acquire a slot from the leaf associated with the local queue. If the leaf is zero, we try to fetch from its parent, continuing upward until we reach the root. If nothing is available at that level, we suspend the queue. If there is unused capacity elsewhere in the tree, queues in that part of the tree will eventually be throttled. Capacity will then percolate upward, and ready queues will be awoken.

When releasing slots (in the completion interrupt handler, when the local queue is throttled or empty), we first choose a queue to awaken. We then release slots to the lowest common ancestor (LCA) of the local and the target CPUs in the token tree. Finally, we awaken the target CPU with an interprocessor interrupt (IPI). The strategy of picking nearby queues tends to keep capacity near the leaves of the token tree, minimizing contention at the higher levels, minimizing the cost of the IPI, and maximizing the likelihood that slots will be passed through a cache line in a higher level of the memory hierarchy. Experiments described in Sec. 4.3.2 confirm that IPIs significantly outperform an alternative based on polling.

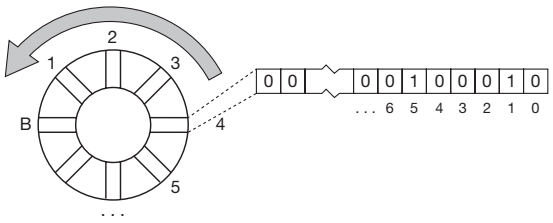

Figure 4: Timer wheel for throttled queues. If queue q is $k > T$ units ahead of global virtual time, it is placed in bucket $\min(\lceil k/b \rceil, B)$, where B is the number of buckets and b is a quantization parameter. In the figure, queues 1 and 5 are throttled in bucket 4.

3.3 Ready and Throttled Queues

The D parameter in MQFQ controls the number of outstanding requests and is a trade-off between utilization and fairness. While a larger D may better utilize the device, it can also impose looser fairness bounds and higher waiting time for incoming requests from a slower flow. Therefore, MQFQ will stop dispatching once there are D outstanding requests in the device. A queue in this case is likely to be both nonempty and unthrottled; such a queue is said to be *ready*.

As noted in Sec. 3.2, a completion handler whose local queue is empty or throttled will give away its released token. To do so, it looks for the closest queue (based on a pre-computed proximity matrix) that is flagged as ready and passes the token through the token tree.

Regardless of the number of outstanding requests, a queue will be throttled when its lead request is T ahead of global virtual time. When this happens, we need to be able to tell when virtual time has advanced enough that the queue can be unthrottled. To support this operation, we employ a simple variant of the classical timer wheel structure [43, 47] (Fig. 4). Each *bucket* of the wheel represents a b-unit interval of virtual time, and contains (as a bitmask) the set of queues that should be unthrottled at the beginning of that interval. Conceptually, we turn the wheel every b time units (in actuality, of course, we update an index that identifies bucket number 1), clear the bitmask in the old bucket 1, and unthrottle the queues that used to appear in that mask.

Given a finite number of buckets, B, a queue that needs to be throttled for longer than $B \times b$ will be placed in bucket B; this means that the wakeup handler for a queue must always double-check to make sure it doesn't have to throttle the queue again. Unlike a classical timing wheel, which contains a list of timer events in every bucket, our bitmask buckets can be manipulated with compare-and-swap, making the whole wheel trivially nonblocking.

As noted in Sec. 3.2, when slots become available in a completion handler, we choose queues from among the ready set, release the slots to the token tree, and send IPIs to the CPUs of the chosen queues. In a similar vein, if slots are available at the root of the token tree when the throttling wheel is turned, we likewise identify ready queues to which to send

IPIs. No fairness pathology arises in always choosing nearby queues: if some far-away queue lags too far behind, nearby queues will end up throttling, slots will percolate upward in the token tree, and the lagging queues will make progress.

3.4 Determining D and T in Practice

In practice, we use a hand-curated workload with varying degrees of concurrency and request sizes (with an approach similar to that of Chen et al. [9]) as a one-time setup step to discover the internal parallelism of a given multi-queue device which determines the parameter D. Any smaller value for D will not saturate the device, while larger Ds would lead to greater unfairness – specially for burstier workloads.

Unlike D which is determined solely by the degree of parallelism in the multi-queue device, the parameter T is affected by the characteristic of the workload – i.e., concurrency and request size. While a single-threaded workload can afford to have $T = 0$, a workload with small requests being submitted from multiple threads across multiple sockets require larger T value. To that end, once we have determined D, in a one-time setup step, we *over-provision* the parameter T for the worst-case workload so that the maximum throughput of the device can always be met.

4 Evaluation

We evaluate fairness and performance of MQFQ on two fast, multi-queue devices: NVMe over RDMA (NVMf, with multi-queue NICs) and multi-queue SSD (MQ-SSD). We also evaluate the scalability of each of our concurrent data structures.

In our NVMf setup, the host machine (where MQFQ runs) issues NVMe requests over RDMA to the target machine, which serves the requests directly from DRAM. We use the kernel host stack and SPDK [21] target stack. This setup can reach nearly 4 M IOP/s for 1KB requests. In our MQ-SSD setup, requests are fulfilled by a PCIe-attached Intel P3700 NVMe MQ-SSD. This setup provides nearly 0.5 M IOP/s for 4K requests.

We measured (with an approach similar to that of Chen et al. [9]) available internal parallelism to be 128 for the NVMf setup and 64 for the MQ-SSD setup. We chose T in each setup to be (roughly) the smallest value that didn't induce significant contention. We preconditioned the MQ-SSD with sequential writes to the whole address space followed by a series of random writes to reach steady state performance. We also disabled power management to ensure consistent results. We ran all experiments on a Linux 4.15 kernel in which KPTI [12] was disabled via boot parameter. For scalability experiments, thread affinities were configured to fill one hardware thread on each core of the first socket, then one on each core of the second socket before returning to populate the second hardware thread of each core. The CPU mask for fairness experiments was configured to partition the cores among competing tasks. Table 1 summarizes the experimental setup. In all of the experiments we use the length of requests in KB to advance virtual time—hence the unit for T is KB. Because the MQ-SSD setup has significantly lower bandwidth than the NVMf setup, we use it only for fairness experiments, not for scalability. The source code for our implementation is available at `http://github.com/hedayati/mqfq`.

4.1 Fairness and Efficiency

We compare MQFQ to two existing systems: (1) the recommended Linux setup for fast parallel devices, which performs no I/O scheduling (`nosched`) and is thus contention free, and (2) Budget Fair Queueing (BFQ) [46], a proportional share scheduler included for multi-queue stacks since Linux 4.12. For each of these, we consider three benchmarks: (a) the Flexible I/O Tester (FIO) [3], (b) the FlashX graph processing framework [52], and (c) the Aerospike key-value store [41].

FIO: FIO is a microbenchmark that allows flexible configuration of I/O patterns and scales quite well. We use FIO to generate workloads with known characteristics. Because FIO does so little processing per request, we also use it as an antagonist in multiprogrammed runs with FlashX and Aerospike. Each FIO workload has a name of the form αxβ (e.g., `2x4K`) where α indicates the number of threads (each on a dedicated queue) and β indicates the size of each request. For proportional sharing tests, we also indicate the weight of the flow in parentheses (e.g., `2x4K(3)`). The FIO queue depth (i.e., the number of submitted but not yet completed requests) is set to 128—large enough to maintain maximum throughput to the device.

To evaluate fairness and efficiency, we consider co-runs of FIO workloads where the internal device scheduler (if any) fails to provide fairness. We compare the slowdown of the flows relative to their time when running alone (in the absence of resource competition) as a measure of fairness as well as aggregated throughput as a measure of efficiency. We explore three cases in which competing flows differ in only one characteristic—request size, concurrency, or priority (weight). The results show that the underlying request processing, being oblivious to these characteristics, fails to provide fairness.

In Fig. 5 top-left and bottom-left, each of the flows uses an equal number of device queues. The device alternates between queues and guarantees the same number of processed requests from each. This results in flows sharing the device in proportion to request sizes rather than getting equal shares.

Fig. 5 top-middle and bottom-middle show two flows, one of which uses half the number of physical queues used by the other flow. With both flows submitting 4KB requests, the requests are processed in proportion to the number of utilized queues, causing unfairness.

Finally, Fig. 5 top-right and bottom-right show how MQFQ can be used to enforce shares in proportion to externally-specified per-flow weights (shown in parentheses).

In all of the above cases, the BFQ scheduler also guarantees fairness (as defined by flows' throughputs) but at a much higher cost compared to MQFQ.

Table 1: Experimental setup.

	MQ-SSD Setup	NVMf Setup
CPU & Mem.	Intel E5-2620 v3 (Haswell) @ 2.40GHz – 8GB	Intel E5-2630 v3 (Haswell) @ 2.40GHz – 64GB
Sockets×Cores	2×6 (24 hardware threads)	2×8 (32 hardware threads)
Target device	Intel P3700 NVMe MQ-SSD (800GB)	NVMe over RDMA Mellanox ConnectX-3 VPI Dual-Port 40Gbps
MQFQ parameters	$D = 64$, $T = 45$KB	$D = 128$, $T = 64$KB

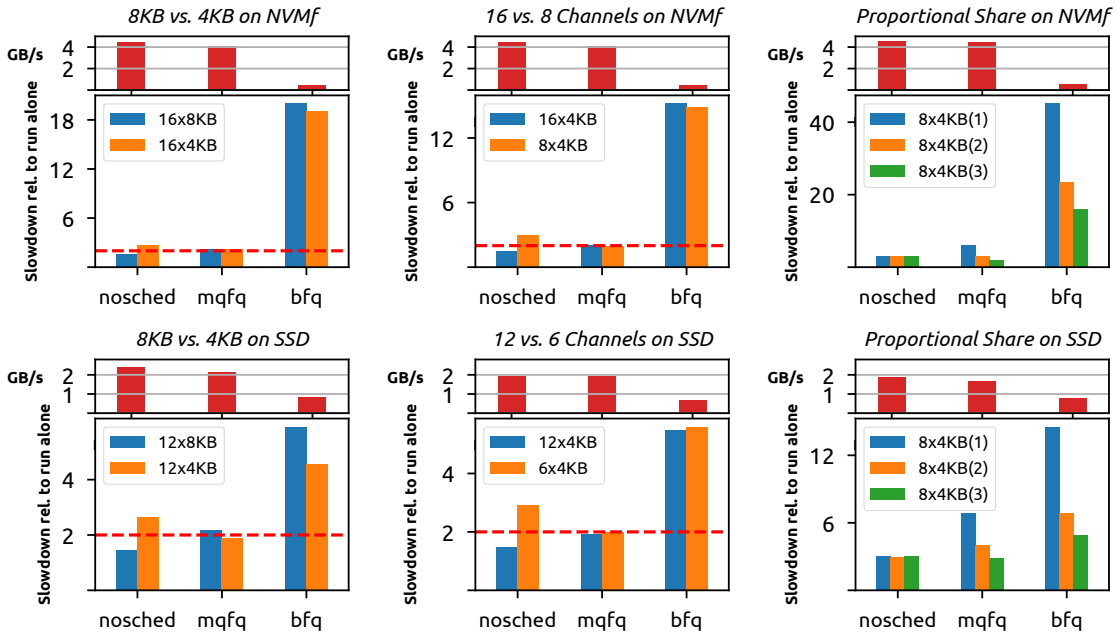

Figure 5: FIO fairness and efficiency. Round-robin (`nosched`) processing is unfair with respect to different request sizes (left), different numbers of queues (middle) and different proportional shares (right). Red dashed lines in the left and middle columns indicate proportional (ideal) slowdown. Aggregate bandwidth is shown above each graph.

FlashX: FlashX is a data analytics framework that utilizes SSDs to scale to large datasets. It can efficiently store and retrieve large graphs and matrices, and uses FlashR, an extended R programming framework, to process terabyte-scale datasets in parallel. We used FlashX to execute pagerank on the SOC-LiveJournal1 social network graph from SNAP [28]. The graph has 4.8M vertices and 68.9M edges and is stored on SSD or the NVMf target's DRAM for corresponding tests. We use FIO as an antagonist process to create contention with FlashX over the storage resource.

Fig. 6 shows the slowdown of co-runs of FlashX and FIO with different schedulers (or none—`nosched`). FlashX does not maintain a large queue depth; as a result, it can sustain only a fraction of the device's throughput. FIO, by contrast, is able to fully utilize the device given its large (I/O) parallelism. Running these together, MQFQ guarantees that FlashX gets its small share of I/O, while the rest is available to FIO, resulting in small (better than proportional) slowdowns (33% for FlashX and 14% for FIO on average between MQ-SSD and NVMf) — note that this is not unexpected since one of the flows, i.e., FlashX, is not saturating. While BFQ also reduces the slowdown for FlashX (from almost 4× to less than 2×), it slows down FIO due to its lack of support for I/O parallelism.

Aerospike: Aerospike is a flash-optimized key-value store. It uses direct I/O to a raw device in order to achieve high performance. Meta-data is kept in memory, but we configure our instance to make sure all requests will result in an I/O to the underlying device. We use the benchmark tool provided with Aerospike, running on a client machine, to drive a workload of small (512B) reads, ensuring that there will be no contention over the network for the NVMf setup. As in the FlashX experiments, we use FIO as a competitor workload.

Fig. 7 shows the slowdown of co-runs of Aerospike and FIO under BFQ, MQFQ, and `nosched`. For the NVMf setup, despite performing nearly 1 M transactions /sec., Aerospike fails to saturate the device before running out of CPUs. Therefore, as with FlashX, the co-run under MQFQ has negligible slowdown (3% for Aerospike and less than 20% for FIO). However, on the MQ-SSD setup Aerospike can fully utilize the

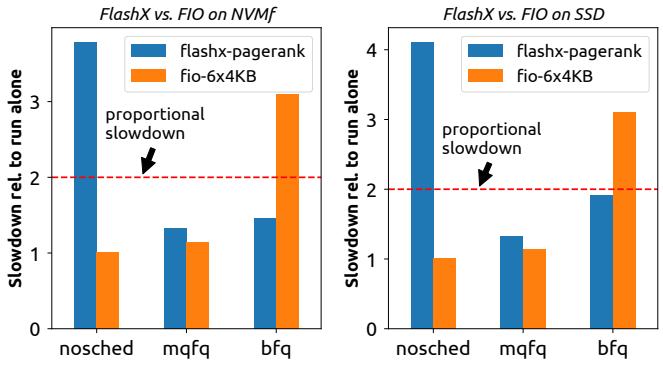

Figure 6: Fairness comparison for FlashX. MQFQ maintains fairness for FlashX, while allowing FIO to utilize the remaining bandwidth of the device.

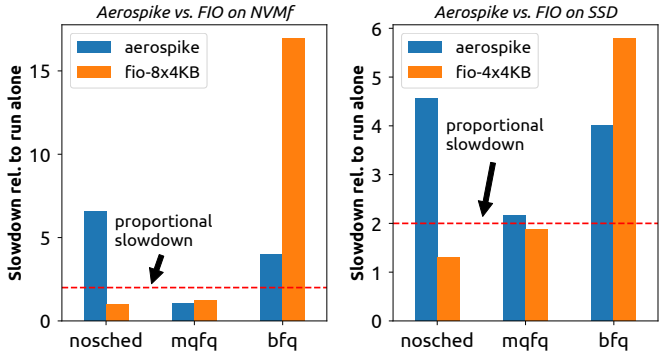

Figure 7: Fairness comparison for Aerospike. MQFQ maintains fairness (approximately at or below proportional slowdown). On the MQ-SSD (right), where Aerospike can utilize the full device, FIO is slowed down to half the available bandwidth.

device (with nearly 0.5 M transactions /sec.) and Aerospike and FIO end up getting half the available bandwidth each. BFQ's lack of support for parallel dispatch is evident on the faster NVMf device, where it results in 15× slowdown for FIO while giving only a modest improvement for Aerospike.

4.2 Scalability

We compare the scalability of MQFQ to that of an existing single-queue implementation of fair queueing—i.e., BFQ [46]. As noted in Sec. 1, Linux BFQ doesn't support concurrent dispatches and may not be able to fully utilize a device with internal parallelism. Other schedulers with support for parallel dispatch (e.g., FlashFQ [40]) have no multi-queue implementation. As a reasonable approximation of the missing strategy, we also compare MQFQ to a modified version of itself (MQFQ-serial) that serializes dispatches using a global lock. It differs from a real single-queue scheduler for a device with internal parallelism in that it maintains the requests in separate, per-CPU queues coordinated with our scalable data structures and the T and D parameters.

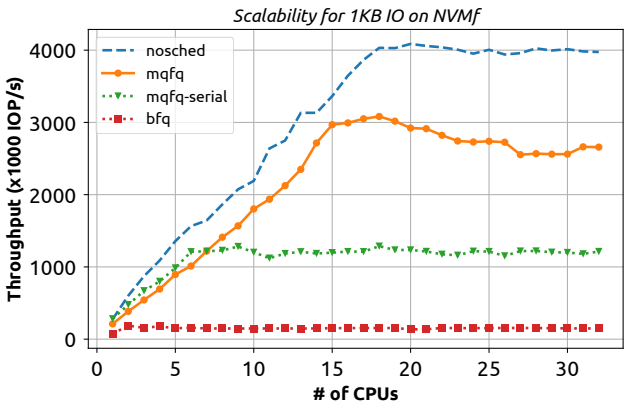

Figure 8: Overall scalability of unfair Linux multi-queue vs. MQFQ vs. MQFQ-serial vs. BFQ.

Our MQ-SSD setup at 460K IOP/s is not suitable for scalability experiments—the IOP/s limit, rather than the scheduler, becomes the scaling bottleneck. Some higher-IOP/s devices exist in the market and more will surely emerge in the future. Employing an array of SSDs can also enable over a million IOP/s. Alternatively, remote storage software solutions (e.g., ReFlex [26], NVMe over Fabric [33], FlashNet [45]) have the potential to yield more than a million IOP/s.

For this scalability evaluation, we therefore rely on the NVMf setup with 1KB requests. We chose 1KB because it yields the largest number of IOP/s (more request churn, leading to higher scheduler contention). In the `nosched` (no contention) case, this setup can reach 4 M IOP/s. We need multiple FIO threads to reach this maximum throughput.

Fig. 8 compares the throughput achieved with `nosched`, MQFQ, MQFQ-serial, and BFQ. With 15–19 active threads, MQFQ reaches more than 3 M IOP/s—2.6× better than MQFQ-serial and 20× better than BFQ. This constitutes 71% of the peak throughput of the in-memory NVMf device while providing the fairness properties needed for shared systems (as demonstrated in Sec. 4.1).

4.3 Design Decisions and Parameters

We assess the degree to which each of MQFQ's scalable data structures improves performance.

4.3.1 Virtual Time

We first evaluate the scalability of computing virtual time in MQFQ. As described in Sec. 3.1, our implementation uses a variant of the mindicator [29] to find the smallest start tag among queued requests across all queues. As in the token tree (Fig. 3), we structure the mindicator with successive levels for cores, sockets, and the full machine.

Fig. 9 shows how the mindicator scales with the number of queues. We are unaware of any existing data structure suitable as a replacement for the mindicator; we therefore implemented another lock-free alternative in which the minimum is

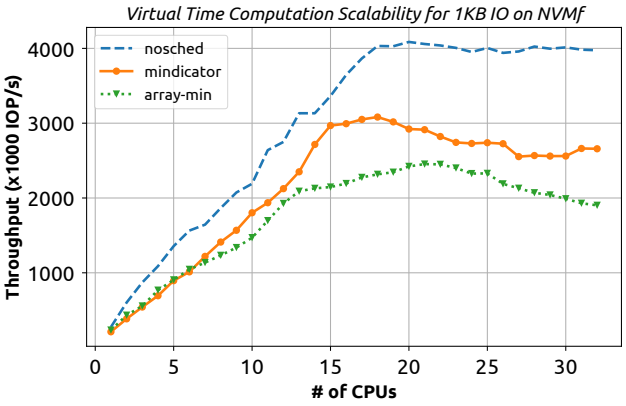

Figure 9: Throughput when maintaining virtual time with a mindicator vs. iterating over an array of queue minima.

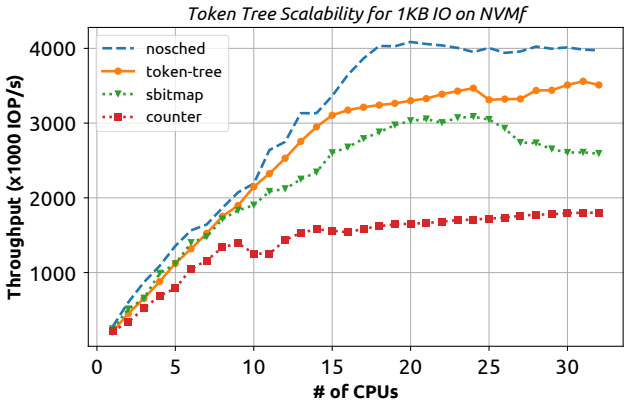

Figure 11: Scalability of token-tree vs. global counter vs. scalable bitmap in maintaining available dispatch slots.

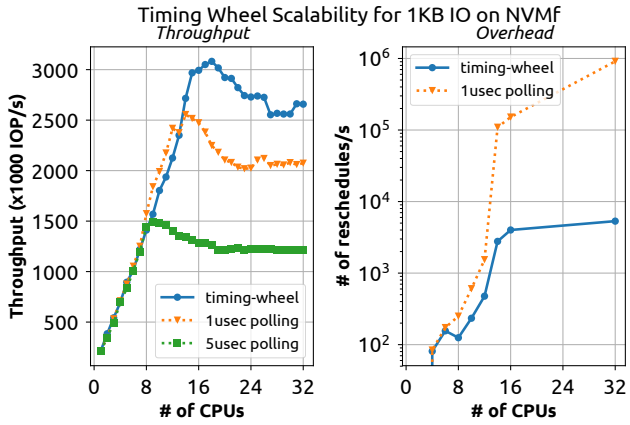

Figure 10: Scalability of unthrottling. Left: MQFQ throughput achieved using timing wheel vs. 1μs and 5μs polling. Right: polling causes spurious reschedules.

found by iterating over an array of queue-local minima after each request dispatch. (This could be thought as a one-level mindicator.) Our contention-localizing structure outperforms the array scan by nearly 40% at 32 threads.

4.3.2 Unthrottling

As discussed in Sec. 2, when a queue cannot dispatch it will be throttled. Once the situation changes (completion or progress of virtual-time) some throttled queues may need to be unthrottled. Any delay in doing so could leave the device underutilized. Our approach uses inter-processor interrupts to promptly notify appropriate CPUs that they can proceed when the unthrottling condition is met. We use a scalable timer wheel (Sec. 3.3) to support such notifications efficiently.

For comparison, arranging for each queue to poll the condition would be an easy but expensive way to implement unthrottling. We explore this option with a pinned, high resolution timer (hrtimer [11]), as it requires no communication between queues and can provide latency comparable to that of a cross-socket inter-processor interrupt. The timer is armed whenever the queue is throttled and upon firing, reschedules the dispatch routine. The effect is essentially polling for a change in virtual time, with a polling frequency determined by the value of the timer.

Fig. 10 (left) compares the throughput that MQFQ can achieve with the timing wheel vs. polling at 1μs or 5μs intervals. Results confirm that a delay in unthrottling leads to throughput degradation. Even extremely frequent (1μs) polling cannot achieve IOP/s performance comparable to that of our timer wheel approach. Less frequent polling leads to a dispatch delay that leaves the device underutilized.

In order to quantify the wasted CPU, we measure the number of reschedule operations caused by our timer wheel and by 1μs polling. The difference between the two shows how inefficient polling can be (the timer wheel incurs no spurious reschedules). Fig. 10 (right) shows the savings, in reschedules per second, achieved by using the timer wheel instead of a 1μs timer. With a few CPUs, roughly every queue is being signaled on every completion (so a carefully chosen frequency for polling that matches the rate of completion could be practical when the device is fully utilized), but the number of wasted cycles grows with the number of CPUs. With the timing wheel, on the other hand, unthrottling comes only as a result of completion, and therefore is upper-bounded by the device throughput.

4.3.3 Dispatch Slots

In order to keep a device with internal parallelism fully utilized, while also avoiding queue build-up in the device (which would adversely affect the fairness guarantee), MQFQ has to track the number of available dispatch slots. This number is modified by each queue as a result of a dispatch or a completion. Our scalable MQFQ design uses a novel token tree data structure for this purpose (presented in Sec. 3.2).

Kyber [39], a multi-queue I/O scheduler added since Linux 4.12, uses another data structure, called *sbitmap* (for Scalable

Bitmap), to throttle asynchronous I/O operations if the latency experienced by synchronous operations exceeds a threshold. The main idea in sbitmap is to distribute the tokens as bits in a number of cache lines (determined by expected contention over acquiring tokens). A thread tries to find the first clear bit in the cache line where the last successful acquire happened, falling back to iterating over all cache lines if all bits are set. This data structure reduces contention when the number of tokens is significantly larger than the number of threads. Yet another alternative to maintain a single global count of available dispatch slots using atomic increments and decrements.

Fig. 11 plots 1KB MQFQ IOP/s as a function of thread count using an atomic counter, a scalable bitmap, and a token tree to track the number of dispatched requests. To isolate the impact of these data structures, we disable virtual time computation in MQFQ. Using an atomic counter doesn't scale beyond the first socket. The scalable bitmap falls short when the number of waiting requests is significantly larger than device parallelism, resulting in local acquire and release of tokens. In comparison, the token tree paired with our throttling mechanism prefers interaction with local queues (based on a pre-computed proximity matrix) as long as they are no more than T ahead of virtual time, resulting in significantly better scalability (more than 2× the throughput of the atomic counter and 36% more than the scalable bitmap).

5 Related Work

Fairness-oriented resource scheduling has been extensively studied in the past. Lottery scheduling [49] achieves probabilistic proportional-share resource allocation. Fairness can also be realized through per-task timeslices as in Linux CFQ [2] and BFQ [46], Argon [48], and FIOS [35]. Timeslice schedulers, however, are generally not *work-conserving*: they will sometimes leave the device unused when there are requests available in the system. The original fair queueing approaches, including Weighted Fair Queueing (WFQ) [13], Packet-by-Packet Generalized Processor Sharing (PGPS) [34], and Start-time Fair Queueing (SFQ) [18], employ virtual-time–controlled request ordering across per-flow request queues to maintain fairness.

Fair queueing approaches like SFQ(D) [23] and FlashFQ [40] have been tailored to manage I/O resources, allowing requests to be re-ordered and dispatched concurrently for better I/O efficiency in devices with internal parallelism. To maintain fairness in a multi-resource (e.g., CPU, memory and NIC) environment, DRFQ [16] adapted fair queueing by tracking usage of the respective dominant resource of each operation. Disengaged fair queueing [30] emulates the effect of fair queueing on GPUs while requiring only infrequent OS kernel involvement. It accomplishes its goal by monitoring and mitigating potential unfairness through occasional traps. All previous fair queueing schedulers assume a serializing scheduler over a single device queue, which does not scale well on modern multicores with fast multi-queue devices.

For multi-queue SSDs, Ahn et al. [1] supported I/O resource sharing by implementing a bandwidth throttler at the Linux cgroup layer (above the multi-queue device I/O paths). However, their time interval budget-based resource control is not work conserving: if one cgroup does not use its allotted resources in an interval, those resources are simply wasted. Lee et al. [27] improved read performance by isolating queues of multi-queue SSDs used for reads from those used for writes. Kyber [39] achieves better synchronous I/O latency by throttling asynchronous requests. However, neither approach is a full solution for fair I/O resource management. Stephens et al. [42] found that the internal round-robin scheduling of hardware queues in NICs leads to unfairness when the load is asymmetrically distributed across a NIC's multiple hardware queues. Their solution, Titan, requires programmable NICs to internally implement deficit round-robin and service queues in proportion to configured weights. FLIN [44] identifies major sources of interference in multi-queue SSDs and implements a scheduler in SSD controller firmware to protect against them. Unlike MQFQ, which is applicable to accelerators and multi-queue NICs, FLIN deals with the idiosyncrasies of Flash devices such as garbage collection and access patterns. In addition, FLIN considers any request originating from the same host-side I/O queue as belonging to the same "flow" and, being implemented in hardware, is unable to reason in terms of system-level resource principals (applications, virtual machines, or Linux cgroups).

For performance isolation and quality-of-service, ReFlex [26] employs a per-tenant token bucket mechanism to achieve latency objectives in a shared-storage environment. The token bucket mechanism and fair queueing resource allocation are complementary—the former performs admission control under a given resource allocation while the latter supports fair, work-conserving resource uses. Decibel [31] presents a system framework for resource isolation in rack-scale storage but it does not directly address the problem of resource scheduling. It uses two existing scheduling policies in its implementation—strict time sharing is not work-conserving; deficit round robin is work-conserving but requires a serializing scheduler queue that limits scalability.

Among multicore operating systems, Arrakis [36] and IX [4] support high-speed I/O by separating the control plane (managed by the OS) and the data plane (bypassing the OS) to achieve coherency-free execution. Their OS control planes enforce access control but not resource isolation or fair resource allocation. Zygos [37] suggests that sweeping simplification introduced by shared-nothing architectures like IX [4] leads to (1) not being work-conserving and (2) suffering from head-of-the-line blocking. They propose a work-stealing packet processing scheme that, while introducing cross-core interactions, eliminates head-of-the-line blocking and improves latency. Recent work has also built scalable data structures that localize synchronization in the multicore memory hierarchy (intra-core rather than inter-core; intra-socket rather

than inter-socket). Examples include the mindicator global minimum data structure [29], atomic broadcast trees [24], and NUMA-aware locks [15] and data structures [7]. For MQFQ, we introduce new scalable structures, including a timer wheel to track virtual time indexes and a token tree to track available device dispatch slots.

6 Conclusion

With the advent of fast devices that can complete a request every microsecond or less, it has become increasingly difficult for the operating system to fulfill its responsibility for fair resource allocation—enough so that some OS implementations have given up on fairness altogether for such devices. Our work demonstrates that surrender is not necessary: with judicious use of scalable data structures and a reformulation of the analytical bounds, we can maintain fairness in the long term and bound it in the short term, all without compromising throughput.

Our formalization of multi-queue fair queueing introduces a parameter, T, that bounds the amount of service that a flow can receive in excess of its share. Crucially, this bound does not grow with time. Moreover, our new definition of virtual time is provably equivalent to existing definitions when T is set to zero. Experiments with a modified Linux 4.15 kernel, a two-socket server, and a fast NVMe over RDMA device confirm that MQFQ can provide both fairness and very high throughput. Compared to running without a fairness algorithm on an NVMf device, our MQFQ algorithm reduces the slowdown caused by an antagonist from 3.78× to 1.33× for the FlashX workload and from 6.57× to 1.03× for the Aerospike workload. Its peak throughput reaches 3.1 Million IOP/s on a single machine, outperforming a serialized version of our own algorithm by 2.6× and Linux BFQ by 20×.

In future work, we plan to develop strategies for automatic tuning of the T and D parameters; extend our implementation to handle small computational kernels for GPUs and accelerators; and evaluate the extent to which fairness guarantees can continue to apply even to kernel-bypass systems, with dispatch queues in user space.

Acknowledgment

We thank our shepherd, Jian Huang, and the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grants CNS-1319417, CCF-1717712, CCF-1422649 and by a Google Faculty Research award. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] S. Ahn, K. La, and J. Kim. Improving I/O resource sharing of Linux cgroup for NVMe SSDs on multi-core systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, Denver, CO, June 2016.

[2] J. Axboe. Linux block IO—Present and future. In *Ottawa Linux Symp.*, pages 51–61, Ottawa, ON, Canada, July 2004.

[3] J. Axboe et al. Flexible I/O tester. github.com/axboe/fio.

[4] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 49–65, Broomfield, CO, Oct. 2014.

[5] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux block IO: Introducing multi-queue SSD access on multi-core systems. In *6th ACM Intl. Systems and Storage Conf. (SYSTOR)*, Haifa, Israel, June 2013.

[6] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *9th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Vancouver, BC, Canada, 2010.

[7] I. Calciu, S. Sen, M. Balakrishnan, and M. K. Aguilera. Black-box concurrent data structures for NUMA architectures. In *22nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 207–221, Xi'an, China, Apr. 2017.

[8] B. Caldwell. Improving block-level efficiency with *scsi-mq*. *arXiv e-prints*, abs/1504.07481v1, Apr. 2015.

[9] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *2011 IEEE 17th Intl. Symp. on High Performance Computer Architecture (HPCA)*, pages 266–277, San Antonio, TX, 2011.

[10] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *24th ACM Symp. on Operating Systems Principles (SOSP)*, pages 1–17, Farminton, PA, 2013.

[11] J. Corbet. The high-resolution timer API. lwn.net/Articles/167897.

[12] J. Corbet. The current state of kernel page-table isolation. lwn.net/Articles/741878/, Dec. 2017.

[13] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 1–12, Austin, TX, Sept. 1989.

[14] D. Dice, Y. Lev, and M. Moir. Scalable Statistics Counters. In *25th ACM Symp. on Parallelism in*

Algorithms and Architectures (SPAA), pages 43–52, Montreal, PQ, Canada, 2013.

[15] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: A general technique for designing NUMA locks. *ACM Trans. on Parallel Compututing*, 1(2):13:1–13:42, Feb. 2015.

[16] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. In *ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 1–12, Helsinki, Finland, 2012.

[17] S. J. Golestani. A self-clocked fair queueing scheme for broadband applications. In *13th IEEE Conf. on Networking for Global Communications (INFOCOM)*, pages 636–646, San Jose, CA, 1994.

[18] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Trans. on Networking*, 5(5):690–704, Oct. 1997.

[19] A. G. Greenberg and N. Madras. How fair is fair queueing. *Journal of the ACM*, 39(3):568–598, July 1992.

[20] Hyper-Q Example. developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf.

[21] Intel Corp. Storage performance development kit. www.spdk.io.

[22] Intel Optane SSD DC P4800X Series. www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series/p4800x-750gb-aic.html.

[23] W. Jin, J. S. Chase, and J. Kaur. Interposed Proportional Sharing for a Storage Service Utility. In *Joint Intl. Conf. on Measurement and Modeling of Computer Systems*, SIGMETRICS, pages 37–48, New York, NY, 2004.

[24] S. Kaestle, R. Achermann, R. Haecki, M. Hoffmann, S. Ramos, and T. Roscoe. Machine-aware atomic broadcast trees for multicores. In *12th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 33–48, Savannah, GA, Nov. 2016.

[25] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *USENIX Annual Technical Conf. (ATC)*, pages 437–450, Denver, CO, June 2016.

[26] A. Klimovic, H. Litz, and C. Kozyrakis. Reflex: Remote flash ≈ local flash. In *22nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 345–359, Xi'an, China, Apr. 2017.

[27] M. Lee, D. H. Kang, M. Lee, and Y. I. Eom. Improving read performance by isolating multiple queues in NVMe SSDs. In *11th Intl. Conf. on Ubiquitous Information Management and Communication*, Beppu, Japan, Jan. 2017.

[28] J. Leskovec and A. Krevl. SNAP datasets: Stanford large network dataset collection. snap.stanford.edu/data/.

[29] Y. Liu, V. Luchangco, and M. Spear. Mindicators: A Scalable Approach to Quiescence. In *2013 IEEE 33rd Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 206–215, Philadelphia, PA, July 2013.

[30] K. Menychtas, K. Shen, and M. L. Scott. Disengaged scheduling for fair, protected access to fast computational accelerators. In *19th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Salt Lake City, UT, Mar. 2014.

[31] M. Nanavati, J. Wires, and A. Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *14th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 17–33, Boston, MA, Mar. 2017.

[32] Nvidia Corp. Sharing a GPU between MPI processes: Multi-process service (MPS). docs.nvidia.com/deploy/mps/index.html.

[33] NVM Express Workgroup. NVM express, revision 1.3a. nvmexpress.org/wp-content/uploads/NVM-Express-1_3a-20171024_ratified.pdf, Oct. 2017.

[34] A. K. Parekh. *A generalized processor sharing approach to flow control in integrated services networks*. PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT, 1992.

[35] S. Park and K. Shen. FIOS: A Fair, Efficient Flash I/O Scheduler. In *10th USENIX Conf. on File and Storage Technologies (FAST)*, pages 13–13, San Jose, CA, 2012.

[36] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the control plane. In *11th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Broomfield, CO, Oct. 2014.

[37] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *26th Symp. on Operating Systems Principles (SOSP)*, pages 325–341, Shanghai, China, 2017.

[38] Samsung SSD PM1725a. www.samsung.com/semiconductor/global/file/insight/2016/08/Samsung_PM1725a-1.pdf.

[39] O. Sandoval. Kyber multi-queue I/O scheduler. lwn.net/Articles/720071/.

[40] K. Shen and S. Park. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In *USENIX Annual Technical Conf. (ATC)*, San Jose, CA, June 2013.

[41] V. Srinivasan, B. Bulkowski, W.-L. Chu, S. Sayyaparaju, A. Gooding, R. Iyer, A. Shinde, and T. Lopatic. Aerospike: Architecture of a real-time

operational dbms. *Proc. of the VLDB Endowment*, 9(13):1389–1400, Sept. 2016.

[42] B. Stephens, A. Singhvi, A. Akella, and M. Swift. Titan: Fair packet scheduling for commodity multiqueue NICs. In *USENIX Annual Technical Conf. (ATC)*, pages 431–444, Santa Clara, CA, 2017.

[43] S. A. Szygenda, C. W. Hemming, and J. M. Hemphill. Time flow mechanisms for use in digital logic simulation. In *5th ACM Winter Simulation Conf.*, pages 488–495, New York, NY, 1971.

[44] A. Tavakkol, M. Sadrosadati, S. Ghose, J. S. Kim, Y. Luo, Y. Wang, N. M. Ghiasi, L. Orosa, J. Gómez-Luna, and O. Mutlu. FLIN: Enabling fairness and enhancing performance in modern NVMe solid state drives. In *45th Intl. Symp. on Computer Architecture (ISCA)*, pages 397–410, Los Angeles, CA, 2018.

[45] A. Trivedi, N. Ioannou, B. Metzler, P. Stuedi, J. Pfefferle, I. Koltsidas, K. Kourtis, and T. R. Gross. Flashnet: Flash/network stack co-design. In *10th ACM Intl. Systems and Storage Conf. (SYSTOR)*, pages 15:1–15:14, Haifa, Israel, 2017.

[46] P. Valente and A. Avanzini. Evolution of the BFQ Storage-I/O scheduler. algo.ing.unimo.it/people/paolo/disk_sched/mst-2015.pdf.

[47] G. Varghese and A. Lauck. Hashed and hierarchical timing wheels: Efficient data structures for implementing a timer facility. *ACM/IEEE Trans. on Networking*, 5(6):824–834, Dec. 1997.

[48] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *5th USENIX Conf. on File and Storage Technologies (FAST)*, pages 61–76, San Jose, CA, Feb. 2007.

[49] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *1st USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 1–11, Monterey, CA, Nov. 1994.

[50] Skyhawk & Skyhawk Ultra NVMe PCIe SSD. www.sandisk.com/content/dam/sandisk-main/en_us/assets/resources/data-sheets/Skyhawk-Series-NVMe-PCIe-SSD-DS.pdf.

[51] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan. Performance analysis of NVMe SSDs and their implication on real world databases. In *8th ACM Intl. Systems and Storage Conf. (SYSTOR)*, Haifa, Israel, May 2015.

[52] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity SSDs. In *13th USENIX Conf. on File and Storage Technologies (FAST)*, pages 45–58, Santa Clara, CA, 2015.