Multi-Queue Fair Queuing *

Technical Report #1005

Mohammad Hedayati,1 Michael L. Scott,1 Kai Shen2 and Mike Marty2

1Department of Computer Science, University of Rochester
2Google Inc.
{hedayati,scott}@cs.rochester.edu
{kshen,mikemarty}@google.com

October 2018

Abstract

Recent high-speed devices (network interfaces, external storage, computational accelerators) provide multiple access channels to allow concurrent I/O from mutually untrusting applications or virtual machines, enabling them to serve millions of ops/s. Unfortunately, it is difficult for the operating system to fairly partition bandwidth among resource principals (VMs, flows, etc.) without introducing synchronization overhead that would largely negate the benefits of multi-channel parallelism.

Existing resource scheduling algorithms, which generally assume underlying serial operation, are also unsuitable to devices with a high degree of internal parallelism. To address these challenges, we present the first known fair, work-conserving scheduler for multi-channel I/O devices. Specifically, we (1) reformulate the classical notion of virtual time to accommodate parallel channels, and (2) describe a scalable implementation that bounds potential unfairness while minimizing synchronization overhead.

Our implementation of multi-queue fair queueing (MQFQ) in the Linux 4.15 kernel demonstrates that it is possible—against all prior expectations—to achieve both fairness and very high throughput. Evaluation with an NVMe over RDMA fabric (NVMf) device demonstrates MQFQ offers up to 20× more throughput and superior fairness compared to an existing Linux fairness algorithm (BFQ)—a rate of up to 3.1 Million IOP/s on a single machine. Compared to running without a fairness algorithm, 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.

1. Introduction

Recent years have witnessed the proliferation of very fast devices for I/O, networking, and computing acceleration. Commodity solid-state disks (e.g., the Intel Optane DC P4800X [23] and Samsung PM1725a [38]) can perform at or near 1 M I/O operations per second. System-area networks (e.g., InfiniBand) can sustain several million remote operations per second over a single link [26]. RDMA delivers data across fabric within several microseconds. GPUs and machine learning accelerators may offload computations that run just a few microseconds at a time [31]. Simultaneously, the proliferation of multicore processors has necessitated architectures tuned for parallel I/O across multiple hardware threads.

It is now common for a high-performance hardware/software stack to embody a multi-queue I/O architecture in which each CPU context (hardware thread) owns a dedicated per-CPU command/completion queue pair, giving it independent access to the device. Examples of this approach include the Windows and Linux NVMe drivers, the Linux multi-queue block layer [5], and SCSI Multi-Queue support [9]. A recent study [51] demonstrated up to 8× performance improvement for YCSB-on-Cassandra on a multi-queue NVMe I/O path compared to single-queue SATA-based I/O.

To support the full bandwidth of modern devices, multi-queue I/O systems are generally designed to avoid any cross-CPU synchronization. In other words, designers have concluded that conventional fair-share I/O schedulers [24, 35], which serialize on a single I/O scheduling queue, are unsuited for modern fast devices.

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.

Unfortunately, by bypassing the OS resource scheduler, direct multi-queue device accesses undermine the OS’s traditional responsibility of resource management and isolation. While I/O devices (e.g., SSD firmware, NICs) may themselves schedule among multiple hardware queues, their support for fairness is hampered by their inability to reason in terms of system-level resource principals (applications, virtual machines, or Linux cgroups) and isolation policies, or to manage an arbitrary number of flows. As a result, device-level schedul-
ing tends to cycle naively among hardware command submission queues in a round robin fashion. Given such simple scheduling, a greedy application or virtual machine may gain unfair resources 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.

Additionally, it is common for modern SSDs [10] and accelerators [21, 40] to internally support parallel requests. Existing resource scheduling algorithms, which assume underlying serial operation, are also unsuitable to devices with a high degree of internal parallelism.

To overcome these problems, we present what we believe to be the first fair queueing scheduler to scalably accommodate multi-queue I/O devices with parallel dispatch. As in classical fair queueing [13, 34], we assure that each flow (in Linux, a cgroup) receives its allocated share of bandwidth. While classical fair queueing employs a single serializing request queue, we observe that the fair queueing principle can be realized on a multi-queue architecture as long as we can efficiently track global resource utilization and arrange to throttle a queue that has exceeded its share by some bounded amount. Accordingly, we introduce the notion of a throttling threshold $T$ such that multiple queues can dispatch in parallel as long as the lead request in each is within $T$ of the resource utilized by the slowest queue, system-wide. 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 scalable implementation of multi-queue fair queueing (MQFQ) is the need for cross-queue synchronization. Against all prior expectations, we demonstrate that it is possible, by choosing appropriate data structures and by leveraging the throttling threshold $T$, to keep the overhead low enough to sustain million-IOPs throughput while satisfying system-wide goals for proportional sharing. An instance of the mindicator of Liu et al. [30] allows us to track flows’ shares without a global cache miss on every I/O completion. A novel data structure we call the token tree allows us to track available device dispatch slots. 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 [44, 47] keeps track of queues whose head requests are too far ahead of the shares of their contributing flows. When the mindicator indicates a sufficient advance in resource utilization of other flows, update of a single index suffices to turn the wheel and enable every queue in the next bucket to proceed.

In summary, our MQFQ algorithm demonstrates that while scalable multicore I/O architectures preclude serialization, they can tolerate infrequent, physically localized synchronization, allowing us to achieve both fairness and high performance.

2. 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 time-slices 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) [19], employ virtual-time-controlled request ordering across per-flow request queues to maintain fairness.

Fair queueing approaches like SFQ(D) [24] and FlashFQ [41] 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 [17] adapted fair queueing by tracking usage of the respective dominant resource of each operation. Disengaged fair queueing [31] 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 channel, which does not scale well on modern multicores with fast multi-queue devices.

There is a growing body of research focusing on exploiting the internal parallelism of SSDs. Chen et al. [10] quantify the internal parallelism of storage medium. Wang et al. [50] show parallelism-aware implementation of a key-value store to perform up to 4× better than one that doesn’t consider internal parallelism of the device. ParaFS [52] is a file system optimized to exploit the internal parallelism of flash devices and reports 3× throughput enhancements in some workloads compared to the state-of-the-art. These systems, while achieving high parallel performance, do not support fair resource management.

For multi-queue NVMe SSDs, Ahn et al. [1] supported I/O resource sharing by implementing a bandwidth throttle at the Linux cgroup layer (above the multi-queue device I/O paths). The bandwidth throttle is adaptively updated in each time interval based on each cgroup container’s budget and a carry-over of unused budget credit from the previous interval. In contrast to fair queueing, time interval budget-based resource control is not work conserving: if one container does not use its allotted resources in an interval, those resources are simply wasted. Lee et al. [28] improved read performance by isolating channels of NVMe 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. [43] found that the internal round-robin scheduling of hardware channels 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.

Recent work has also focused on performance isolation and quality-of-service on fast storage services in data centers. ReFlex [27] employs a per-tenant token bucket mechanism to guarantee latency objectives in a shared-storage environment. The token bucket mechanism and fair queuing resource allocation are complementary—the former performs admission control under a given resource allocation while the latter supports fair, work-conserving resource uses. Decibel [32] 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.

In multithreaded operating system designs, Corey [6] argued for application-controlled sharing to enhance scalability while Multikernel [3] proposed the separation of per-CPU kernels to minimize sharing and contention. 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 model. 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 introduce a work-stealing packet processing scheme that, while introduces 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 [30], atomic broadcast trees [25], and NUMA-aware locks [15] and data structures [8]. 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.

3. Design

3.1. Background: Classical Fair Queueing

Fair queuing [13, 34] is a class of algorithms to schedule a shared (network, processing, or I/O) resource among competing flows. Each flow consists of a sequence of requests or packets arriving at the router or server. Each request has an associated cost, which reflects its resource utilization (e.g., service time or bandwidth). Fair queuing allocates the capacity of the resource in proportion to weights assigned to the competing flows (only the relative values of the weights are significant).

A flow is said to be active if it has one or more outstanding requests, and backlogged if it has outstanding requests that have not yet been dispatched (made available to the device). Fair queuing algorithms are work-conserving: they schedule requests of active flows to consume surplus resources in proportion to the weights of the active flows. A flow whose requests arrive too slowly to maintain a backlog may forfeit the unconsumed portion of its share.

Start-time Fair Queuing SFQ [19, 20] 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 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 idle, virtual time is defined to be equal to the maximum finish tag of any request that has been serviced by that time.

Parallel Dispatch A server with internal parallelism may service multiple requests simultaneously, so virtual time is not well-defined for conventional SFQ 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) [24] works the same as SFQ but allows up to D in-service requests (D = 1 reduces to SFQ). Due to out-of-order completion, virtual time is computed either as the minimum of outstanding (not yet dispatched) requests or the maximum of dispatched (being processed in the device) requests.

3.2. Multi-Queue Fair Queuing

The main obstacle in adapting fair queuing—or most other scheduling algorithms, for that matter—to a multi-queue I/O architecture is the apparent need to order requests through a single shared 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 completion events, since these may occur out of order.

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
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 and described in Sec. 4) to bound unfairness across flows.

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 $T$ to zero, while limits scalability in practice, would effectively restore the semantics of a global priority queue. Setting it greater than zero leads to relaxed semantics but lower synchronization overhead. Specifically, it allows short-term fluctuations of as much as $T$ in the relative progress of flows, while preserving long-term shares. By adjusting $T$ appropriately, we can hope to find a design point that provides most of the fairness of traditional fair queuing with most of the performance of fully independent queues.

For an internally parallel device, we will often need to dispatch a new request before the previous one has finished, in order to keep the device busy. At the same time, since the device decides the order in which dispatched requests are served, we must generally avoid dispatching more requests than what can actually be handled in parallel, thereby preserving our ability to order these held-back requests. We therefore introduce a second tuning 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 hold idle resources back from being allotted to other flows, leading to underutilization. To fix this, a parallelism-aware scheduler needs to use backlogged (instead of active) flows to determine virtual time. The virtual time (and thus the start tag of a newly arriving request on a previously non-backlogged flow) is defined to be the minimum start tag of all requests across all queues. In a multi-queue system, computing this global minimum without frequently incurring cache misses is challenging. In Sec. 4 we show how we localize the misses using a mindicator [30].

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. This relaxation of fairness precludes the use of any of the approaches discussed by Jin et al. [24], where the only challenge is out-of-order completion.

We now define our notion of per-flow virtual time, in a way that accommodates concurrency in 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 flow $f$ 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 is 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. Significantly, this equivalence allows our implementation 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. Our MQFQ algorithm 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.

### 3.3. 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 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:

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

(1)

Clearly, this is possible only if the flows can be broken into infinitesimal units. For a packet- or block-based resource
global structures:
- VT mindicatoer
- 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
- 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).

We similarly derive bounds on the fairness achieved by MQFQ. Our analysis builds on the fairness bounds for Start-time Fair Queueing (SFQ) [19] and SFQ(D) [24]. Goyal et al. [19] 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:

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

is twice the lower bound. SFQ uses a single priority queue and serves one request at a time. Now, we consider the fairness bounds for 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 + $T$ (throttling threshold). 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:

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

We provide a proof sketch for 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 $T$:

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

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 $l_f^{\text{max}}$, the finish tag of the last completed request must be at least $v_2 - T - l_f^{\text{max}} / r_f$. Therefore if we just count the completed requests in $[t_1, t_2]$, the minimum service received by backlogged flow $f$ is at least $r_f \cdot (v_2 - T - v_1) - l_f^{\text{max}}$.

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

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

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 $l_f^{\text{max}}$ may be outstanding and, in the maximum case, almost entirely serviced. Counting the completed requests and the
As described in Secs. 1 and 3.2, virtual time in MQFQ reflects
we prefer to dispatch from the local queue, queues on the same
As discussed in Sec. 3.2, MQFQ employs a separate priority
vides the same fairness bound as SFQ(D).

Time, in turn, is the minimum start tag of any request
misses to only the source and target queues, it would also

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

\[ \frac{W_f(t_1, t_2)}{r_f} - \frac{W_m(t_1, t_2)}{r_m} \leq 2T + \frac{f_{\text{max}}}{r_f} + \frac{m_{\text{max}}}{r_m} \]

This completes the proof of Theorem 1.

Note that when the throttling threshold T = 0, MQFQ(1)
provides the same fairness bound as SFQ. Therefore the throt-
ling threshold represents a tradeoff between fairness and scal-
able implementation in a multicore 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 back-
logged 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:

\[ \frac{W_f(t_1, t_2)}{r_f} - \frac{W_m(t_1, t_2)}{r_m} \leq (D + 1) \left( 2T + \frac{f_{\text{max}}}{r_f} + \frac{m_{\text{max}}}{r_m} \right) \]

This is true based on a combination of Theorem 1 and the
proved fairness bound for SFQ(D) [24]. We omit the detailed
proof. When the throttling threshold T = 0, MQFQ(D) pro-
vides the same fairness bound as SFQ(D).

**4. Scalability**

As discussed in Sec. 3.2, MQFQ employs a separate priority
queue for every CPU (hardware thread), to minimize coher-
ence 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 possi-
able. 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.

**4.1. Virtual Time**

As described in Secs. 1 and 3.2, 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 the
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 maintaining 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. This operation, however, is far
too expensive to perform on a regular basis (see Sec. 5.3.1).

Instead, we use an instance of Liu et al.’s mindicator
structure [30], modified to preclude decreases in the global min-
imum. 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 chil-
dren’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 re-
quests in a highly parallel device complete at least slightly out
of virtual-time 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 values, updated (with fetch-and-add) on each
dispatch of a request. In future work, we plan to explore
whether better performance might be obtained with a scalable
monotonic counter [7,14], at least for flows with many threads.

**4.2. Available Slots**

A queue in MQFQ is unable to dispatch either when it is too
far ahead of virtual time or because the device is currently
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 device can handle in parallel, thereby avoid
queue buildup in the device and preserve our ability to choose
the ideal request to submit when an outstanding request com-
pletes.

Our experiments show 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 (see Sec. 5.3.3). We
therefore aim to improve locality by preferentially allocating
available slots to physically nearby queues, in a manner remi-
niscent of cohort locks [15]. This approach meshes well with
our notification mechanism, which prefers to unblock nearby
queues.

Assume as a result of a completion we have slots available
for use by a queue (other than the current one). Instead of
updating a global counter, we could pass the available slots
as tokens to the target queue. While this approach limits the
misses to only the source and target queues, it would also
4.3. Ready and Throttled Queues

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. 4.2, a completion handler whose local queue is throttled or empty, we first choose a value of all nodes together sum to the difference between nonempty and unthrottled; such a queue is said to be ready.

When releasing slots (in the completion interrupt handler, when the local queue is throttled or empty), we first choose 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 awaken.

When releasing slots (in the completion interrupt handler, when the local queue is throttled or empty), we first choose 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 awakened.

As an alternative to sending an IPI, we could awaken queues by setting a flag that would be polled by the owner CPU using timer interrupts. Since completions on a fast device can occur every microsecond, however, we would need to poll at very high frequency in order to use available slots as quickly as possible. Given that timer interrupts are not dramatically faster than IPIs, we have chosen the interrupt-based approach in our current implementation (see Sec. 5.3.2).

5. Evaluation

We evaluate fairness guarantees and performance of MQFQ on two fast, multi-queue devices: NVMe over RDMA (NVMf) and SSD. We also evaluate the scalability of each of our data structures (mindicator for maintaining virtual time, timer wheel for unthrottling, and token tree for maintaining the number requests that are being processed in parallel).
Table 1: Experimental setups.

<table>
<thead>
<tr>
<th></th>
<th>SSD Setup</th>
<th>NVMf Setup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU &amp; Mem.</strong></td>
<td>Intel E5-2620 v3 (Haswell) @ 2.40GHz – 8GB</td>
<td>Intel E5-2630 v3 (Haswell) @ 2.40GHz – 64GB</td>
</tr>
<tr>
<td><strong>Sockets × Cores</strong></td>
<td>2×6 (24 hardware threads)</td>
<td>2×8 (32 hardware threads)</td>
</tr>
<tr>
<td><strong>Target device</strong></td>
<td>Intel P3700 NVMe SSD (800GB)</td>
<td>NVMe over RDMA</td>
</tr>
<tr>
<td><strong>MQFQ parameters</strong></td>
<td>( D = 64, T = 45\text{KB} )</td>
<td>( D = 128, T = 64\text{KB} )</td>
</tr>
</tbody>
</table>

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 [22] target stack. This setup can reach nearly 4 M IOP/s for 1KB requests. In our SSD setup, requests are fulfilled by a PCIe-attached Intel P3700 NVMe SSD. This setup provides nearly 0.5 M IOP/s for 4K requests. We 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 setups. 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 SSD setup has significantly lower bandwidth than the NVMf setup, we use it only for fairness experiments, not for scalability.

### 5.1. 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 IO Tester (FIO) [16], (b) the FlashX graph processing framework [53], and (c) the Aerospike key-value store [42].

**FIO:** FIO is a microbenchmark that allows flexible configuration of I/O patterns and scales up quite well. Each FIO workload has a name of the form \( \alpha \times \beta \) (e.g., \( 2 \times 4\text{K} \)) where \( \alpha \) indicates the number of threads (each on a dedicated queue) and \( \beta \) indicates the size of each request. For proportional sharing tests, we also indicate the weight of the flow in parentheses (e.g., \( 2 \times 4\text{K} \times 3 \)). 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 the fairness and efficiency of MQFQ, 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 channels. The device alternates between channels and guarantees the same number of processed requests from each channel. This translates into flows sharing device in proportion to their request sizes as opposed to getting equal shares.

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

Finally, Fig. 5 top-right and bottom-right show how MQFQ can be used to enforce shares proportional 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.

**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 utilizes FlashR, an extended R programming framework, to process datasets at a scale of terabytes in parallel. We used FlashX to execute pagerank on the SOC-LiveJournal1 social network graph from SNAP [29]. 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 (and no scheduler for nosched). FlashX does not maintain a large queue depth for the requests that it issues; 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 (IO) parallelism. Running these together, MQFQ guarantees that FlashX gets its small share of

---
Figure 5: FIO fairness and efficiency. Round-robin (nosched) processing is unfair with respect to different request sizes (left), different numbers of channels (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.

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

IO, while the rest is available to FIO, resulting in small (better than proportional) slowdowns (33% for FlashX and 14% for FIO on average between SSD and NVMf). While BFQ also reduces the slowdown for FlashX (from almost $4 \times$ to less than $2 \times$), it slows down FIO due to its lack of support for IO parallelism.

**Aerospike:** Aerospike is a flash-optimized key-value store. It uses direct IO to a raw device in order to achieve high performance. Meta-data is still kept in memory, but we configure our instance to make sure all requests will result in an IO 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 no scheduler. For the NVMf setup, despite performing nearly 1 M transactions per second, Aerospike fails to saturate the device before running out of dedicated 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 SSD setup Aerospike can fully utilize the device (with nearly 0.5 M transactions/sec.) and the co-run results in Aerospike and FIO getting half the available bandwidth each. BFQ’s lack of support for parallel IO dispatch is evident on the faster NVMf device, where it results in $15 \times$ slowdown for FIO while giving only a modest improvement for Aerospike.

5.2. 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 [41]) 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.

Our 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, also employing an array of SSDs can provide over a million IOP/s. Alternatively, remote storage software solutions (e.g., ReFlex [27], NVMe over Fabric [33], FlashNet [45]) have been introduced that promise over 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 throughput compared to MQFQ-serial and 20× better than BFQ. This achieves 71% the peak throughput of the in-memory NVMf device while providing the valuable fairness properties needed for shared systems (as demonstrated in Section 5.1).

5.3. Design Decisions and Parameters

We assess how each of MQFQ’s scalable data structures improves performance.

5.3.1. Virtual Time We first evaluate the scalability of computing virtual time in MQFQ. As described in Sec. 4.1, our implementation uses a variant of the mindicator [30] 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 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.

5.3.2. Unthrottling As discussed in Sec. 3, 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 CPU that they can proceed when the unthrottling condition is met. We use a scalable timer wheel (Sec. 4.3) 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 with frequencies of 1 μs and 5 μs. The results confirm that a delay in unthrottling leads to throughput degradation. Even extremely frequent (1 μs) polling cannot achieve IOP/s performance comparable to that of our timer wheel approach. Less frequent polling leads to a delay of dispatch 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 1 μs 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 1 μs timer. While 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), 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 throughput of the device.

5.3.3. 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. 4.2).

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 IO 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 cachelines (determined by expected contention over acquiring tokens). A thread tries to find the first clear bit in the cacheline where the last successful acquire happened, falling back to iterating over all cachelines 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. In 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 unthrottling 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 $2 \times$ the throughput of the atomic counter and 36% more than the scalable bitmap).

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 allocation of resources—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 queuing 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 NVMe device, our MQFQ algorithm reduces the slowdown caused by an antagonist from 3.78x to 1.33x for the FlashX workload and from 6.57x to 1.03x for the Airspike workload. Its peak throughput reaches 3.1 Million IOP/s on an NVMf device.  

References


