BFQ (Budget Fair Queueing)

BFQ is a proportional-share I/O scheduler, with some extra low-latency capabilities. In addition to cgroups support (blkio or io controllers), BFQ's main features are:

- BFQ guarantees a high system and application responsiveness, and a low latency for time-sensitive applications, such as audio or video players;
- BFQ distributes bandwidth, and not just time, among processes or groups (switching back to time distribution when needed to keep throughput high).

In its default configuration, BFQ privileges latency over throughput. So, when needed for achieving a lower latency, BFQ builds schedules that may lead to a lower throughput. If your main or only goal, for a given device, is to achieve the maximum-possible throughput at all times, then do switch off all low-latency heuristics for that device, by setting low_latency to 0. See Section 3 for details on how to configure BFQ for the desired tradeoff between latency and throughput, or on how to maximize throughput.

As every I/O scheduler, BFQ adds some overhead to per-I/O-request processing. To give an idea of this overhead, the total, single-lock-protected, per-request processing time of BFQ---i.e., the sum of the execution times of the request insertion, dispatch and completion hooks---is, e.g., 1.9 us on an Intel Core i7-2760QM@2.40GHz (dated CPU for notebooks; time measured with simple code instrumentation, and using the throughput-sync.sh script of the S suite [1], in performance-profiling mode). To put this result into context, the total, single-lock-protected, per-request execution time of the lightest I/O scheduler available in blk-mq, mq-deadline, is 0.7 us (mq-deadline is ~800 LOC, against ~10500 LOC for BFQ).

Scheduling overhead further limits the maximum IOPS that a CPU can process (already limited by the execution of the rest of the I/O stack). To give an idea of the limits with BFQ, on slow or average CPUs, here are, first, the limits of BFQ for three different CPUs, on, respectively, an average laptop, an old desktop, and a cheap embedded system, in case full hierarchical support is enabled (i.e., CONFIG_BFQ_GROUP_IOSCHED is set), but CONFIG_BFQ_CGROUP_DEBUG is not set (Section 4-2): - Intel i7-4850HQ: 400 KIOPS - AMD A8-3850: 250 KIOPS - ARM CortexTM-A53 Octa-core: 80 KIOPS

If CONFIG_BFQ_CGROUP_DEBUG is set (and of course full hierarchical support is enabled), then the sustainable throughput with BFQ decreases, because all blkio.bfq* statistics are created and updated (Section 4-2). For BFQ, this leads to the following maximum sustainable throughputs, on the same systems as above: - Intel i7-4850HQ: 310 KIOPS - AMD A8-3850: 200 KIOPS - ARM CortexTM-A53 Octa-core: 56 KIOPS

BFQ works for multi-queue devices too.

1. When may BFQ be useful?

BFQ provides the following benefits on personal and server systems.

1-1 Personal systems

Low latency for interactive applications

Regardless of the actual background workload, BFQ guarantees that, for interactive tasks, the storage device is virtually as responsive as if it was idle. For example, even if one or more of the following background workloads are being executed:

- one or more large files are being read, written or copied,
- a tree of source files is being compiled,
- one or more virtual machines are performing I/O,
- a software update is in progress,
- indexing daemons are scanning filesystems and updating their databases,

starting an application or loading a file from within an application takes about the same time as if the storage device was idle. As a comparison, with CFQ, NOOP or DEADLINE, and in the same conditions, applications experience high latencies, or even become unresponsive until the background workload terminates (also on SSDs).

Low latency for soft real-time applications

Also soft real-time applications, such as audio and video players/streamers, enjoy a low latency and a low drop rate, regardless of the background I/O workload. As a consequence, these applications do not suffer from almost any glitch due to the background workload.

Higher speed for code-development tasks

If some additional workload happens to be executed in parallel, then BFQ executes the I/O-related components of typical code-development tasks (compilation, checkout, merge, ...) much more quickly than CFQ, NOOP or DEADLINE.

High throughput

On hard disks, BFQ achieves up to 30% higher throughput than CFQ, and up to 150% higher throughput than DEADLINE and

NOOP, with all the sequential workloads considered in our tests. With random workloads, and with all the workloads on flash-based devices, BFQ achieves, instead, about the same throughput as the other schedulers.

Strong fairness, bandwidth and delay guarantees

BFQ distributes the device throughput, and not just the device time, among I/O-bound applications in proportion their weights, with any workload and regardless of the device parameters. From these bandwidth guarantees, it is possible to compute tight per-I/O-request delay guarantees by a simple formula. If not configured for strict service guarantees, BFQ switches to time-based resource sharing (only) for applications that would otherwise cause a throughput loss.

1-2 Server systems

Most benefits for server systems follow from the same service properties as above. In particular, regardless of whether additional, possibly heavy workloads are being served, BFQ guarantees:

- audio and video-streaming with zero or very low jitter and drop rate;
- fast retrieval of WEB pages and embedded objects;
- real-time recording of data in live-dumping applications (e.g., packet logging);
- responsiveness in local and remote access to a server.

2. How does BFQ work?

BFQ is a proportional-share I/O scheduler, whose general structure, plus a lot of code, are borrowed from CFQ.

- Each process doing I/O on a device is associated with a weight and a *(bfq_)queue*.
- BFQ grants exclusive access to the device, for a while, to one queue (process) at a time, and implements this service model by associating every queue with a budget, measured in number of sectors.
    - After a queue is granted access to the device, the budget of the queue is decremented, on each request dispatch, by the size of the request.
    - The in-service queue is expired, i.e., its service is suspended, only if one of the following events occurs: 1) the queue finishes its budget, 2) the queue empties, 3) a "budget timeout" fires.
        - The budget timeout prevents processes doing random I/O from holding the device for too long and dramatically reducing throughput.
        - Actually, as in CFQ, a queue associated with a process issuing sync requests may not be expired immediately when it empties. In contrast, BFQ may idle the device for a short time interval, giving the process the chance to go on being served if it issues a new request in time. Device idling typically boosts the throughput on rotational devices and on non-queueing flash-based devices, if processes do synchronous and sequential I/O. In addition, under BFQ, device idling is also instrumental in guaranteeing the desired throughput fraction to processes issuing sync requests (see the description of the slice_idle tunable in this document, or [1, 2], for more details).
            - With respect to idling for service guarantees, if several processes are competing for the device at the same time, but all processes and groups have the same weight, then BFQ guarantees the expected throughput distribution without ever idling the device. Throughput is thus as high as possible in this common scenario.
        - On flash-based storage with internal queueing of commands (typically NCQ), device idling happens to be always detrimental for throughput. So, with these devices, BFQ performs idling only when strictly needed for service guarantees, i.e., for guaranteeing low latency or fairness. In these cases, overall throughput may be sub-optimal. No solution currently exists to provide both strong service guarantees and optimal throughput on devices with internal queueing.
    - If low-latency mode is enabled (default configuration), BFQ executes some special heuristics to detect interactive and soft real-time applications (e.g., video or audio players/streamers), and to reduce their latency. The most important action taken to achieve this goal is to give to the queues associated with these applications more than their fair share of the device throughput. For brevity, we call just "weight-raising" the whole sets of actions taken by BFQ to privilege these queues. In particular, BFQ provides a milder form of weight-raising for interactive applications, and a stronger form for soft real-time applications.
    - BFQ automatically deactivates idling for queues born in a burst of queue creations. In fact, these queues are usually associated with the processes of applications and services that benefit mostly from a high throughput. Examples are systemd during boot, or git grep.
    - As CFQ, BFQ merges queues performing interleaved I/O, i.e., performing random I/O that becomes mostly sequential if merged. Differently from CFQ, BFQ achieves this goal with a more reactive mechanism, called Early Queue Merge (EQM). EQM is so responsive in detecting interleaved I/O (cooperating processes), that it enables BFQ to achieve a high throughput, by queue merging, even for queues for which CFQ needs a different mechanism, preemption, to get a high throughput. As such EQM is a unified mechanism to achieve a high throughput with interleaved I/O.
    - Queues are scheduled according to a variant of WF2Q+, named B-WF2Q+, and implemented using an augmented rb-tree to preserve an O(log N) overall complexity. See [2] for more details. B-WF2Q+ is also ready for hierarchical

scheduling, details in Section 4.

- B-WF2Q+ guarantees a tight deviation with respect to an ideal, perfectly fair, and smooth service. In particular, B-WF2Q+ guarantees that each queue receives a fraction of the device throughput proportional to its weight, even if the throughput fluctuates, and regardless of: the device parameters, the current workload and the budgets assigned to the queue.
- The last, budget-independence, property (although probably counterintuitive in the first place) is definitely beneficial, for the following reasons:
  - First, with any proportional-share scheduler, the maximum deviation with respect to an ideal service is proportional to the maximum budget (slice) assigned to queues. As a consequence, BFQ can keep this deviation tight not only because of the accurate service of B-WF2Q+, but also because BFQ *does not* need to assign a larger budget to a queue to let the queue receive a higher fraction of the device throughput.
  - Second, BFQ is free to choose, for every process (queue), the budget that best fits the needs of the process, or best leverages the I/O pattern of the process. In particular, BFQ updates queue budgets with a simple feedback-loop algorithm that allows a high throughput to be achieved, while still providing tight latency guarantees to time-sensitive applications. When the in-service queue expires, this algorithm computes the next budget of the queue so as to:
    - Let large budgets be eventually assigned to the queues associated with I/O-bound applications performing sequential I/O: in fact, the longer these applications are served once got access to the device, the higher the throughput is.
    - Let small budgets be eventually assigned to the queues associated with time-sensitive applications (which typically perform sporadic and short I/O), because, the smaller the budget assigned to a queue waiting for service is, the sooner B-WF2Q+ will serve that queue (Subsec 3.3 in [2]).

- If several processes are competing for the device at the same time, but all processes and groups have the same weight, then BFQ guarantees the expected throughput distribution without ever idling the device. It uses preemption instead. Throughput is then much higher in this common scenario.
- ioprio classes are served in strict priority order, i.e., lower-priority queues are not served as long as there are higher-priority queues. Among queues in the same class, the bandwidth is distributed in proportion to the weight of each queue. A very thin extra bandwidth is however guaranteed to the Idle class, to prevent it from starving.

3. What are BFQ's tunables and how to properly configure BFQ?

Most BFQ tunables affect service guarantees (basically latency and fairness) and throughput. For full details on how to choose the desired tradeoff between service guarantees and throughput, see the parameters slice_idle, strict_guarantees and low_latency. For details on how to maximise throughput, see slice_idle, timeout_sync and max_budget. The other performance-related parameters have been inherited from, and have been preserved mostly for compatibility with CFQ. So far, no performance improvement has been reported after changing the latter parameters in BFQ.

In particular, the tunables back_seek-max, back_seek_penalty, fifo_expire_async and fifo_expire_sync below are the same as in CFQ. Their description is just copied from that for CFQ. Some considerations in the description of slice_idle are copied from CFQ too.

per-process ioprio and weight

Unless the cgroups interface is used (see "4. BFQ group scheduling"), weights can be assigned to processes only indirectly, through I/O priorities, and according to the relation: weight = (IOPRIO_BE_NR - ioprio) * 10.

Beware that, if low-latency is set, then BFQ automatically raises the weight of the queues associated with interactive and soft real-time applications. Unset this tunable if you need/want to control weights.

slice_idle

This parameter specifies how long BFQ should idle for next I/O request, when certain sync BFQ queues become empty. By default slice_idle is a non-zero value. Idling has a double purpose: boosting throughput and making sure that the desired throughput distribution is respected (see the description of how BFQ works, and, if needed, the papers referred there).

As for throughput, idling can be very helpful on highly seeky media like single spindle SATA/SAS disks where we can cut down on overall number of seeks and see improved throughput.

Setting slice_idle to 0 will remove all the idling on queues and one should see an overall improved throughput on faster storage devices like multiple SATA/SAS disks in hardware RAID configuration, as well as flash-based storage with internal command queueing (and parallelism).

So depending on storage and workload, it might be useful to set slice_idle=0. In general for SATA/SAS disks and software RAID of SATA/SAS disks keeping slice_idle enabled should be useful. For any configurations where there are multiple spindles behind single LUN (Host based hardware RAID controller or for storage arrays), or with flash-based fast storage, setting slice_idle=0 might end up in better throughput and acceptable latencies.

Idling is however necessary to have service guarantees enforced in case of differentiated weights or differentiated I/O-request lengths. To see why, suppose that a given BFQ queue A must get several I/O requests served for each request served for another queue B. Idling ensures that, if A makes a new I/O request slightly after becoming empty, then no request of B is dispatched in the middle, and thus A does not lose the possibility to get more than one request dispatched before the next request of B is dispatched. Note that idling guarantees the desired differentiated treatment of queues only in terms of I/O-request dispatches. To guarantee that the actual service order then corresponds to the dispatch order, the strict_guarantees tunable must be set too.

There is an important flipside for idling: apart from the above cases where it is beneficial also for throughput, idling can severely impact throughput. One important case is random workload. Because of this issue, BFQ tends to avoid idling as much as possible, when it is not beneficial also for throughput (as detailed in Section 2). As a consequence of this behavior, and of further issues described for the strict_guarantees tunable, short-term service guarantees may be occasionally violated. And, in some cases, these guarantees may be more important than guaranteeing maximum throughput. For example, in video playing/streaming, a very low drop rate may be more important than maximum throughput. In these cases, consider setting the strict_guarantees parameter.

slice_idle_us

Controls the same tuning parameter as slice_idle, but in microseconds. Either tunable can be used to set idling behavior. Afterwards, the other tunable will reflect the newly set value in sysfs.

strict_guarantees

If this parameter is set (default: unset), then BFQ

- always performs idling when the in-service queue becomes empty;
- forces the device to serve one I/O request at a time, by dispatching a new request only if there is no outstanding request.

In the presence of differentiated weights or I/O-request sizes, both the above conditions are needed to guarantee that every BFQ queue receives its allotted share of the bandwidth. The first condition is needed for the reasons explained in the description of the slice_idle tunable. The second condition is needed because all modern storage devices reorder internally-queued requests, which may trivially break the service guarantees enforced by the I/O scheduler.

Setting strict_guarantees may evidently affect throughput.

back_seek_max

This specifies, given in Kbytes, the maximum "distance" for backward seeking. The distance is the amount of space from the current head location to the sectors that are backward in terms of distance.

This parameter allows the scheduler to anticipate requests in the "backward" direction and consider them as being the "next" if they are within this distance from the current head location.

back_seek_penalty

This parameter is used to compute the cost of backward seeking. If the backward distance of request is just 1/back_seek_penalty from a "front" request, then the seeking cost of two requests is considered equivalent.

So scheduler will not bias toward one or the other request (otherwise scheduler will bias toward front request). Default value of back_seek_penalty is 2.

fifo_expire_async

This parameter is used to set the timeout of asynchronous requests. Default value of this is 250ms.

fifo_expire_sync

This parameter is used to set the timeout of synchronous requests. Default value of this is 125ms. In case to favor synchronous requests over asynchronous one, this value should be decreased relative to fifo_expire_async.

low_latency

This parameter is used to enable/disable BFQ's low latency mode. By default, low latency mode is enabled. If enabled, interactive and soft real-time applications are privileged and experience a lower latency, as explained in more detail in the description of how BFQ works.

DISABLE this mode if you need full control on bandwidth distribution. In fact, if it is enabled, then BFQ automatically increases the bandwidth share of privileged applications, as the main means to guarantee a lower latency to them.

In addition, as already highlighted at the beginning of this document, DISABLE this mode if your only goal is to achieve a high throughput. In fact, privileging the I/O of some application over the rest may entail a lower throughput. To achieve the highest-possible throughput on a non-rotational device, setting slice_idle to 0 may be needed too (at the cost of giving up any strong guarantee on fairness and low latency).

timeout_sync

Maximum amount of device time that can be given to a task (queue) once it has been selected for service. On devices with costly seeks, increasing this time usually increases maximum throughput. On the opposite end, increasing this time coarsens the granularity of the short-term bandwidth and latency guarantees, especially if the following parameter is set to zero.

max_budget

Maximum amount of service, measured in sectors, that can be provided to a BFQ queue once it is set in service (of course within the limits of the above timeout). According to what said in the description of the algorithm, larger values increase the throughput in proportion to the percentage of sequential I/O requests issued. The price of larger values is that they coarsen the granularity of short-term bandwidth and latency guarantees.

The default value is 0, which enables auto-tuning: BFQ sets max_budget to the maximum number of sectors that can be served during timeout_sync, according to the estimated peak rate.

For specific devices, some users have occasionally reported to have reached a higher throughput by setting max_budget explicitly, i.e., by setting max_budget to a higher value than 0. In particular, they have set max_budget to higher values than those to which BFQ would have set it with auto-tuning. An alternative way to achieve this goal is to just increase the value of timeout_sync, leaving max_budget equal to 0.

4. Group scheduling with BFQ

BFQ supports both cgroups-v1 and cgroups-v2 io controllers, namely blkio and io. In particular, BFQ supports weight-based proportional share. To activate cgroups support, set BFQ_GROUP_IOSCHED.

4-1 Service guarantees provided

With BFQ, proportional share means true proportional share of the device bandwidth, according to group weights. For example, a group with weight 200 gets twice the bandwidth, and not just twice the time, of a group with weight 100.

BFQ supports hierarchies (group trees) of any depth. Bandwidth is distributed among groups and processes in the expected way: for each group, the children of the group share the whole bandwidth of the group in proportion to their weights. In particular, this implies that, for each leaf group, every process of the group receives the same share of the whole group bandwidth, unless the ioprio of the process is modified.

The resource-sharing guarantee for a group may partially or totally switch from bandwidth to time, if providing bandwidth guarantees to the group lowers the throughput too much. This switch occurs on a per-process basis: if a process of a leaf group causes throughput loss if served in such a way to receive its share of the bandwidth, then BFQ switches back to just time-based proportional share for that process.

4-2 Interface

To get proportional sharing of bandwidth with BFQ for a given device, BFQ must of course be the active scheduler for that device.

Within each group directory, the names of the files associated with BFQ-specific cgroup parameters and stats begin with the "bfq." prefix. So, with cgroups-v1 or cgroups-v2, the full prefix for BFQ-specific files is "blkio.bfq." or "io.bfq." For example, the group parameter to set the weight of a group with BFQ is blkio.bfq.weight or io.bfq.weight.

As for cgroups-v1 (blkio controller), the exact set of stat files created, and kept up-to-date by bfq, depends on whether CONFIG_BFQ_CGROUP_DEBUG is set. If it is set, then bfq creates all the stat files documented in Documentation/admin-guide/cgroup-v1/blkio-controller.rst. If, instead, CONFIG_BFQ_CGROUP_DEBUG is not set, then bfq creates only the files:

```
blkio.bfq.io_service_bytes
blkio.bfq.io_service_bytes_recursive
blkio.bfq.io_serviced
blkio.bfq.io_serviced_recursive
```

The value of CONFIG_BFQ_CGROUP_DEBUG greatly influences the maximum throughput sustainable with bfq, because updating the blkio.bfq.* stats is rather costly, especially for some of the stats enabled by CONFIG_BFQ_CGROUP_DEBUG.

Parameters

For each group, the following parameters can be set:

weight

This specifies the default weight for the cgroup inside its parent. Available values: 1..1000 (default: 100).

For cgroup v1, it is set by writing the value to *blkio.bfq.weight*.

For cgroup v2, it is set by writing the value to *io.bfq.weight*. (with an optional prefix of *default* and a space).

The linear mapping between ioprio and weights, described at the beginning of the tunable section, is still valid, but all weights higher than IOPRIO_BE_NR*10 are mapped to ioprio 0.

Recall that, if low-latency is set, then BFQ automatically raises the weight of the queues associated with

interactive and soft real-time applications. Unset this tunable if you need/want to control weights.

weight_device

This specifies a per-device weight for the cgroup. The syntax is *minor:major weight*. A weight of *0* may be used to reset to the default weight.

For cgroup v1, it is set by writing the value to *blkio.bfq.weight_device*.

For cgroup v2, the file name is *io.bfq.weight*.

[1]

P. Valente, A. Avanzini, "Evolution of the BFQ Storage I/O Scheduler", Proceedings of the First Workshop on Mobile System Technologies (MST-2015), May 2015.

http://algogroup.unimore.it/people/paolo/disk_sched/mst-2015.pdf

[2]

P. Valente and M. Andreolini, "Improving Application Responsiveness with the BFQ Disk I/O Scheduler", Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR '12), June 2012.

Slightly extended version:

http://algogroup.unimore.it/people/paolo/disk_sched/bfq-v1-suite-results.pdf

[3]

https://github.com/Algodev-github/S