BFQ I/O scheduler
More throughput, control and efficiency

Paolo Valente <paolo.valente@linaro.org>
In this presentation

- Results are much better than those in the abstract!
- Background: benefits already provided by BFQ
  - Five times more throughput than other solutions for controlling I/O bandwidths
  - Unprecedented responsiveness
- Background: things to improve
  - Mechanisms providing the above benefits become a hindrance or get confused with certain workloads
  - And add extra execution time
- First enhancement(s), following from improvements on above mechanisms:
  - 150% more throughput
  - 80% higher responsiveness (shorter application start-up times)
- Second enhancement
  - 10% lower execution time
Last achievements of BFQ

- In server-like scenarios, BFQ is extremely effective in reaching a high throughput
  - Five times more throughput, while providing much stronger I/O-bandwidth guarantees: [https://www.linaro.org/blog/io-bandwidth-management-for-production-quality-services/](https://www.linaro.org/blog/io-bandwidth-management-for-production-quality-services/)
- BFQ guarantees incomparable responsiveness on Chromebooks too: [https://youtu.be/w2bREYTe0-0](https://youtu.be/w2bREYTe0-0)
  - This convinced the ChromiumOS crew to switch to BFQ: [https://chromium-review.googlesource.com/c/chromiumos/overlays/chromiumos-overlay/+/1440381](https://chromium-review.googlesource.com/c/chromiumos/overlays/chromiumos-overlay/+/1440381)
- Linux 5.0 HDD I/O Scheduler Benchmarks - BFQ Takes The Cake
- Endless OS Switched To The BFQ I/O Scheduler For More Responsive Linux Desktop
- Good news may arrive from Android, SUSE, Fedora and other systems in the near future
Issues

● Goals of an I/O scheduler
  ○ Boost throughput, while controlling I/O, and, yes, run in a very short time
● Very hard to attain all three goals in all scenarios
  ○ The very mechanisms guaranteeing the very high throughput mentioned in previous
    slide cause throughput losses with some scenarios!
    ■ They get confused with some deceptive workloads, and lower throughput for nothing
  ○ On the opposite end, the current tuning for throughput sacrifices responsiveness in some
    scenario
  ○ BFQ is heavier than the other I/O scheduler
    ■ We didn’t even know how much heavier exactly
● From last connect, hard work to improve on every goal
● Let’s start with throughput improvements
● To this goal, let’s see a little bit of internals
General I/O scheduler scheme

Legend:
- Paths followed by operations and requests, from issuing to dispatching
- Components that do I/O operations or move I/O requests along the paths
Serving one request at a time lowers speed

- $T_P$: processing time to dispatch a new I/O request to the drive (OS latency)
- $T_S$: service time of an I/O request in the drive
Then have a new request queued in time!

- I.e., before next completion
- Simplest case: one request served at a time
  - Queue depth 2 is sufficient for full speed if $T_p \leq T_s$
- The depth needed for full speed grows if more reqs are served in parallel
But in-drive queueing causes loss of control...

- E.g., suppose queue $Q_1$ must have two reqs served for each req of queue $Q_2$
- But $Q_1$ issues sync I/O, while $Q_2$ does async I/O with very large depth
Effective solution

- Serve **only one internal queue** (bfq-queue) at a time, for a while
- If a sync bfq-queue remains empty while in service
  - **Do not switch** to another queue for a short time
    - Instead, wait for a new request to arrive for the bfq-queue in service
  - IOW, plug the I/O coming from other bfq-queues, and wait trustingly ...
- **Problem**
  - Typically only one core at a time can execute the code to enqueue requests into a bfq-queue
  - A single core may easily be slower to insert requests into a bfq-queue, than a drive to serve the same requests
  - IOW, $T_P > T_S$ may hold while only one sync bfq-queue is in service
- **Real-world example**
  - CPU: Intel Core i7-2760QM@2.40GHz
  - SSD: PLEXTOR PX-256M5 SATA SSD, with $T_S = 14$ usec
  - $T_P$ while a single sync bfq-queue is served: 200 usec
But, if $T_P > T_S$, then there are service holes

"Drive with internal queueing, single-req service; I/O depth 2, $T_P > T_S$"
Main countermeasure and new commit

- Improve the dropping of extra I/O in service holes
  - Commit: tune service injection basing on request service times
- Since commit “inject other-queue I/O into seeky idle queues on NCQ flash”, BFQ does the following to boost throughput with random I/O on flash storage with internal queueing
Drop **controlled** extra service from other queues

*Drive with internal queueing, single-req service; I/O depth 2, $T_P > T_s$*
Limitations of the current mechanism

- The hard part is finding the right amount of I/O to inject, so as to both boost throughput and not break bandwidth and latency guarantees.
- The current version of this mechanism measures the bandwidth enjoyed by the victim bfq-queue while it is being served, and tries to inject the maximum possible amount of extra service that does not cause that bandwidth to decrease too much.
- But, for measurements to be stable, the queue must be in service for long enough.
- This does not hold in many cases.
- This is one of the reasons why injection is limited to only the less delicate case: only random I/O
  - Largest holes and shortest per-request service times.
The new mechanism

- Changes (only) the way how the amount of injection allowed is dynamically computed
- Tunes injection as a function of the service times of single I/O requests of the victim queue, instead of the bandwidth of the queue
- Single-request service times are evidently meaningful even if a queue gets very few I/O requests completed while it is in service
- Relying on the precision of the new mechanism, injection is now extended also to non-random I/O
Throughput boost with these improvements

- Hard workload for BFQ
  - dbench benchmark
  - Very deceptive I/O for BFQ heuristics
  - Used Phoronix version of this benchmark
- Worst case: six clients
- Device: PLEXTOR PX-256M5 SATA SSD
- This performance boost follows also from two other commits
  - Do not plug I/O for lowest-weight queues
  - Disable queue merging on SSDs
The other two main improvements

- Improved responsiveness
- Reduced execution time
Improving responsiveness

- We have seen that plugging I/O when sync bfq-queues remain empty causes throughput losses.
- For this reason, the waiting time for new requests, and the consequent plugging, are kept short.
  - Current default: 8 ms.
- Unfortunately, such a low value may cause a violation of bandwidth guarantees for a process that happens to issue new I/O too late.
  - The higher the system load, the higher the probability.
- This is a problem when service guarantees matter more than throughput.
  - Important case: weight-raised bfq-queues, which need to be granted a very low latency and therefore a very high bandwidth.
- Simple solution: increase waiting for weight-raised bfq-queues.
- Reassured about throughput by the extra-I/O injection.
Improved responsiveness: example of results

- Device: PLEXTOR PX-256M5 SATA SSD
- Kyber is the non-BFQ I/O scheduler reaching the lowest start-up times
Reducing execution time

- Processes doing sync interleaved I/O: set of processes whose individual sync I/O is random, but whose merged cumulative I/O is sequential
- To boost throughput with these processes
  - BFQ redirects their I/O into a common, shared bfq-queue (queue merging)
  - Thanks to reordering based on request positions, the I/O in this bfq-queue is sequential
- This costs time!
- And does not provide benefits on flash storage with internal queueing, because these devices
  - Enqueue many I/O requests
  - Reorder requests so as to maximise throughput
  - Or simply serve requests in parallel

so they reach the same throughput regardless of BFQ reordering
- Then just don’t do queue merging on these devices!
  - This also boosts throughput by reducing workload asymmetries and need of I/O plugging
Shorter execution time without queue merging

- **MQ_DEADLINE** is the lightest I/O scheduler in blk-mq
  - 800 LOC against 10500 LOC of BFQ
- This total time is the sum of the execution times over three request-processing events
  - Enqueue, dispatch, completion
- So the amortized cost of BFQ, per event, is now ~0.6 usec
  - ~0.4 usec more than MQ-DEADLINE
- This further reduces CPU/drive configurations for which in-kernel I/O handling is still feasible, while BFQ is not
Thank you

Join Linaro to accelerate deployment of your Arm-based solutions through collaboration

contactus@linaro.org