BFQ, fairness and low latency in block I/O
The Two Towers
Paolo Valente
Contents

● Why BFQ?
● What happened since the last episode?
● What is still to come?
  ○ The two Towers
● How is it going in general with latency and fairness?
What are we talking about?

- **I/O Scheduling**: deciding the order in which I/O requests are to be served
  - Especially if several processes compete for the same storage device
- **Order chosen so as to guarantee**:
  - High I/O throughput
  - Low latency
  - High responsiveness
  - Fairness

- I/O scheduling is performed by ad-hoc components in the block layer
blk-mq and I/O schedulers
Why BFQ

- In \textit{blk-mq}, there is now a new I/O Scheduler: BFQ
- Why yet another scheduler?
- Recap with our last demo on a HiKey
  - \url{https://youtu.be/ANfqNiJVoVE}
BFQ upstreamed in April 2017

- After several years
- Merged from 4.12, end of April
- In blk-mq

- What happened since?
Progress summary

- 34 Linux commits upstreamed (16 mine)
- 5 patches waiting for approval (4 ours, see next item)
- Task force of four students set up
- One new module and one new test upstreamed for the Phoronix Test Suite
- First third-party evaluation of BFQ benefits finally available
  - On Phoronix
- BFQ resubmitted for android-hikey-linaro-4.9 too
- Several kernel developers, plus some enthusiastic users, have contributed by testing, reviewing or submitting patches

- Let’s see in detail
BFQ task force

Luca Miccio

Angelo Ruocco

Mirko Montanari

Daniele Toschi
Upstreaming finalization

- 25 of the commits (9 mine) were devoted to finalization and maintenance
  - Mainly bug fixing
- Plus a patch to disable writeback throttling also with BFQ
  - Made by Luca Miccio
  - First patch for mainline by a student of mine!
- One of the bugs was caused by a general issue between blk-mq I/O schedulers and cgroups
  - My fix provided an example of how to address this issue in any scheduler
- Only one of the bugs reported still pending
  - Yet not reported/reproduced any more
Other contributions in a moment

- The other commits and work-in-progress patches are related to the near future of BFQ
- We postpone their description to the part of this presentation on the next challenges
BFQ for Android

- BFQ won’t be available in Android for a long time
  - BFQ is available only from Linux 4.12, in blk-mq
  - It will take time till Android moves to Linux 4.12
- To fill this gap, we prepared and submitted a legacy blk version of BFQ for `android-hikey-linaro-4.9`
- It might be a first step to let BFQ enter Google AOSP
- But such an upstreaming is unlikely to be easy
  - Google Android crew feel that possible regressions might be more relevant than possible benefits
- This resistance is probably related to the following problem
Current vision of BFQ 1/2

- **Best case**
  - “BFQ loads programs extremely faster” (only case backed up by numbers)
  - “... BFQ on a multi-disk server has made it much more responsive.”
  - “BFQ has greatly helped to have a responsive system during [heavy background I/O] and ... never ... any interruption of the video streams”
  - “does seem to make using spinning rust a bit more pleasant” [Mike Galbraith]
  - “It seems to have cured an interactivity issue … Can't really say for sure given this is not based on measurement” [Mike Galbraith]

- **Worst case**
  - “Got anything besides anecdotal evidence?”
  - “BFQ like BFS seems like software based on snake oil placebo effects instead of any real world performance gains”
  - “Do you have independent benchmarks to back this up? I have yet to see any speedups or reduction in latency by using BFQ over CFQ”
Current idea of BFQ 2/2

- Undecided case
  - “And thanks CK for your work.” [email to me from an enthusiastic user 😊]

- Opinion most frequently expressed by block-layer developers
  - “” (empty string)

- So, is BFQ an I/O scheduler with super powers, or just snake oil?
- How are so discordant opinions possible?
BFQ folklore

● Probably for two main reasons
● People still know little about the actual benefits of BFQ
● For those who know these benefits, they know them
  ○ just as folklore
  ○ or because of their impressions (no measurements)

● Repeatable tests are available in my S suite
  ○ But S is very little popular, if not almost completely unknown
  ○ Nobody is using S to confirm or deny my public results

● In the end, the actual impact of BFQ is not well-known
From folklore to sound knowledge

- To address these issues, we tried to have BFQ
  - properly and measurably tested, in terms of its target goals
  - by a third party
  - on a very popular, authoritative benchmark suite

- So, we contributed to the Phoronix Test Suite
  - Module that helps users configure BFQ correctly for the target tests
  - New test, measuring the start-up time of popular applications in the presence of background workload

- Using these contributions, Michael Larabel already wrote an article (report) on Phoronix, for a very fast SSD
- A companion article for an HDD is expected soon
Phoronix results 1/2

- The report does show BFQ benefits
- But is also highlights **latency** and **throughput issues**
- For one of the test cases, latency issues (bad responsiveness) were caused by a combination of four subtle bugs
- Bugs already found and fixed with the help of BFQ task force
  - Four patches prepared and submitted for mainline
  - Angelo Ruocco has co-signed two of the patches
Phoronix results 2/2

- The other issues were all due to BFQ not complying with the characteristics of the storage device
  - BFQ is not yet effective in guaranteeing a high throughput in all scenarios with very fast, queueing storage devices
  - BFQ is not yet fast enough (in terms of execution time) to comply with the request-completion times of modern, very fast devices

- Motivation for the two challenges that BFQ has to face
The Two Towers

- Improve effectiveness
  - Make BFQ effective in providing both high throughput and strong service guarantees (latency, fairness) in any working condition

- Improve efficiency
  - Reduce BFQ execution overhead to such an extent that it complies even with very fast flash-based devices
First contributions to effectiveness

Three commit series already upstreamed

1. Improve handling of flash-based devices with no command queueing
   - Significant throughput boost for random I/O
   - On a HiKey board, throughput increased by up to 125%, growing, e.g., from 6.9MB/s to 15.6MB/s with two or three random readers in parallel

2. Improve throughput indirectly
   - Added instructions, in official documentation (in kernel tree), on how to configure BFQ to get max performance depending on target tasks and devices

3. Improve responsiveness with heavy sync-write workloads in the background
   - `gnome-terminal` start-up time decreased from 50 to 8 seconds on an HDD (against 5 seconds with no background workload)
Contributions waiting for approval

- The 4-patch series containing fixes for the responsiveness issues highlighted by one of the test cases considered in the Phoronix article
First tower

- Going beyond these initial contributions is rather more challenging
- In order to provide strong service guarantees, full control on service order is needed
- Guaranteeing both full control and, at the same time, high throughput is very hard with modern devices
- Let’s see why
The synchronous I/O problem 1/2

- Consider a process, with a very high weight, that shares storage with other very-low-weight processes
- Virtually all the storage-device bandwidth must be devoted to the high-weight process
  - Almost every time the high-weight process has an I/O request, that request must be served immediately
- Suppose that
  - The high-weight process does synchronous I/O: after each I/O request, it systematically blocks, waiting for request completion, then issues a new I/O request
  - The other processes always have some pending I/O request
The synchronous I/O problem 2/2

- What to do when the high-weight process blocks waiting for the completion of its current I/O request?
- If the I/O scheduler dispatches I/O requests from other processes, to not leave the device idle
  - A lot of low-weight I/O may then happen to be queued in the device while the high-weight process is blocked
  - When the high-weight process finally issues a new I/O request, it will find the device busy
  - The new high-weight I/O request may not be served immediately as it had to be, but only after the already-queued low-weight I/O

- Control on the service order of high-weight and low-weight I/O may be completely lost
  - Bandwidth control may be completely lost
Device idling and epic failure

- A simple, clean is solution is just idling the device while waiting for next I/O request for a high-weight synchronous process
- Effective solution for devices without internal queueing
- Nefarious for devices with internal queueing
  - Idling the device means making its internal queue(s) empty
  - **Severe loss of throughput** with a modern device!
- Paradoxical failure for fairness
  - The device can become so underutilized with device idling that a process gets much **less throughput** if treated fairly (with idling) than if treated unfairly (without idling)!
So what?

- Working hard for a solution to have the best of both worlds
  - Or at least enough good from both worlds
- We have some ideas
- Expectedly, it will be about choosing the right trade off …
- The contributions of Adam Manzanares on I/O-request priorities may come in handy
  - Limited to devices honouring these priorities
Second tower

- Improving efficiency is the most difficult task
- BFQ is
  - Large (~9K LOC)
  - Complex
  - Designed with efficiency in mind: unlikely to hide trivial inefficiencies
- Anyway, already implementing some improvements
  - Reducing the average number of steps for updating the I/O schedule
- Major improvements will involve
  - Non-trivial profiling
  - Non-trivial redesign
Latency and fairness landscape

- Last part of this presentation
- Overview of the state of affairs with latency and fairness in storage I/O
  - Ambiguity in the very word latency
  - Unfair fairness
  - The case of Android lag
Latency ambiguity

- Ambiguity on the very word latency
- For technicians, \textit{average} overall processing time of an I/O request
  - Measured in the \textit{absence of long queueing} in the OS
- For users, waiting time for tasks to be accomplished
  - E.g., start an app, get next video frame
  - \textit{Worst case} much more important than average
  - Problems caused \textit{exactly by long queueing}!
- A proper full name for user-perceived latency might be \textit{worst-case I/O-task latency}
  - Abbrev.: \textit{task latency}?
- The term \textit{lag} is already commonly used for task latency if task is interactive
Paradoxical failure for fairness

- Enforcing bandwidth control is needed also to guarantee a low task latency
  - The I/O of the task that must be guaranteed a low latency must be guaranteed a high fraction of the bandwidth
- Idling is needed to control bandwidth
  - Throughput may then be lost
- A user may be willing to sacrifice throughput when needed for low task latency
- Things change dramatically for fairness
  - With device idling, a process may get less throughput if treated fairly than if not!
Fairness without idling 1/2

- Only throttling currently available
  - Simple
  - Efficient
- Yet rigid, indirect and non-trivial to configure
  - To let a process get its intended share of the throughput, limit **the other processes**
    - Need to guess the right limits for the other processes, as a function of the expected throughput reached by the device
    - But overall throughput varies with the I/O-dispatch pattern
    - And throttling influences the I/O-dispatch pattern!
- So, either be conservative to make sure the intended share is guaranteed
  - Underutilize available bandwidth
- Or play at your own risk!
Fairness without idling 2/2

- Even worse, **even in a risky configuration**, throughput may however be lost
  - If some process does I/O below its bandwidth limit, other throttled processes **cannot compensate** for the unused throughput!

- Throttling extensions now available in Linux to reduce this bandwidth waste
  - Try to detect if the bandwidth limit of some process can be raised without degrading latency and bandwidth for other processes

- But with the following shortcomings
  - Very heuristic
  - Effectiveness largely dependent on and variable with workloads
  - Not easy to configure well
  - Unlikely to get optimal device utilization with generic, dynamic workloads
Waiting for a complete solution ...

- Solving throughput problems of proportional-share schedulers like BFQ could finally provide a sound, complete solution
  - No need for complex configurations
  - Seamless redistribution of excess bandwidth
  - Throughput saturation (or quasi saturation) at all times
  - Tight task latencies guaranteed
  - And, of course, fairness
The case of Android lag

- All our experiments show serious lag issues related to I/O
  - E.g., during app updates
- We see the same issues on our phones in daily usage
  - Especially during app updates
  - We did not trace phones 24/7 to completely close the loop
- Users complain publicly
- It seems obvious that there are these lag issues
  - If there is bad or no control on storage access, and some app/service are in an I/O-bound phase, then some other app/service may wait long before having the opportunity to use storage
- Yet, Android seems to suffer from some lag in solving this lag problem
  - Solution held up mainly by the concern that the needed kernel changes may cause regressions or instability
Thank You

#SFO17
SFO17 keynotes and videos on: connect.linaro.org
For further information: www.linaro.org