The two new I/O controllers and *BFQ*

Paolo Valente

University of Modena and Reggio Emilia - Italy
Work sponsored by Linaro

February 24, 2020



Roadmap

First part: state of affairs

- Two new I/O controllers landed in Linux
- Comparisons with *BFQ* show that, with some common workloads, both controllers fail to
  - control bandwidth or latency
  - reach a high throughput

Second part (body): *why*?

- In-depth analysis of why the new controllers fail

Before the two new I/O controllers

Use case

- Multiple entities compete for shared storage
  - Processes, groups, containers, virtual machines, ...

Goal

- Guarantee to each entity a bounded I/O latency and/or at least a minimum I/O bandwidth

- Current solutions waste up to 90% of drives' speed
  `https://www.linaro.org/blog/io-bandwidth-management-for-production-quality-services/`
- The *BFQ* I/O scheduler reduces speed waste to virtually 0%
- Yet *BFQ* is still all but a common solution

The two new *cgroups* I/O controllers

io.latency controller

- Associates a target latency with each group
- Throttles groups so as to meet target latency

io.cost controller

- Associates a weight with each group
- Throttles groups so as let each group get a fraction of the bandwidth proportional to its weight, and inversely proportional to the cost (time) of its I/O
  - This service scheme allows latency to be guaranteed indirectly

And the old *BFQ* I/O scheduler

BFQ I/O scheduler

- Associates a weight with each group or process
- Schedules I/O so as to let each group or process get a fraction of the bandwidth proportional to its weight
  - Slow I/O gets guarantees in the time domain
  - As with *io.cost*, latency guaranteed indirectly

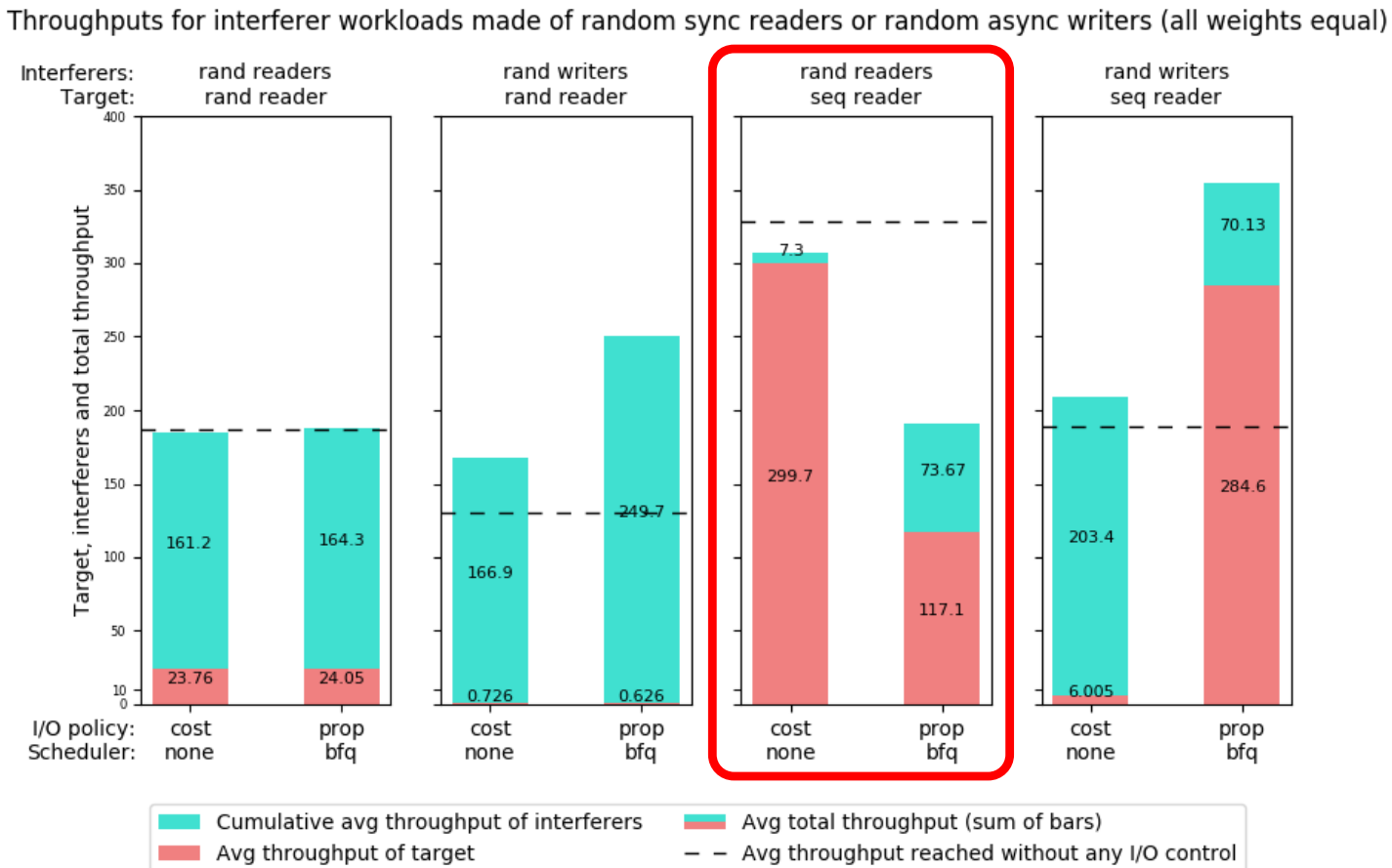
Comparison between controllers and *BFQ*

- The goals of the new controllers are the same as *BFQ*
- No documentation on when to use these controllers and when *BFQ*
- Motivated a comparison between these controllers and *BFQ*: `http://connect.linaro.org/resources/san19/san19-114`
  - Ability to guarantee bandwidth and latency to groups, and to reach a high (total) throughput
- Essential result: *BFQ* outperformed these controllers
  - E.g., on an SSD, *BFQ* proved able to guarantee up to 47 times more bandwidth than *io.cost* to the group under test, and up to 2.8 times more total throughput

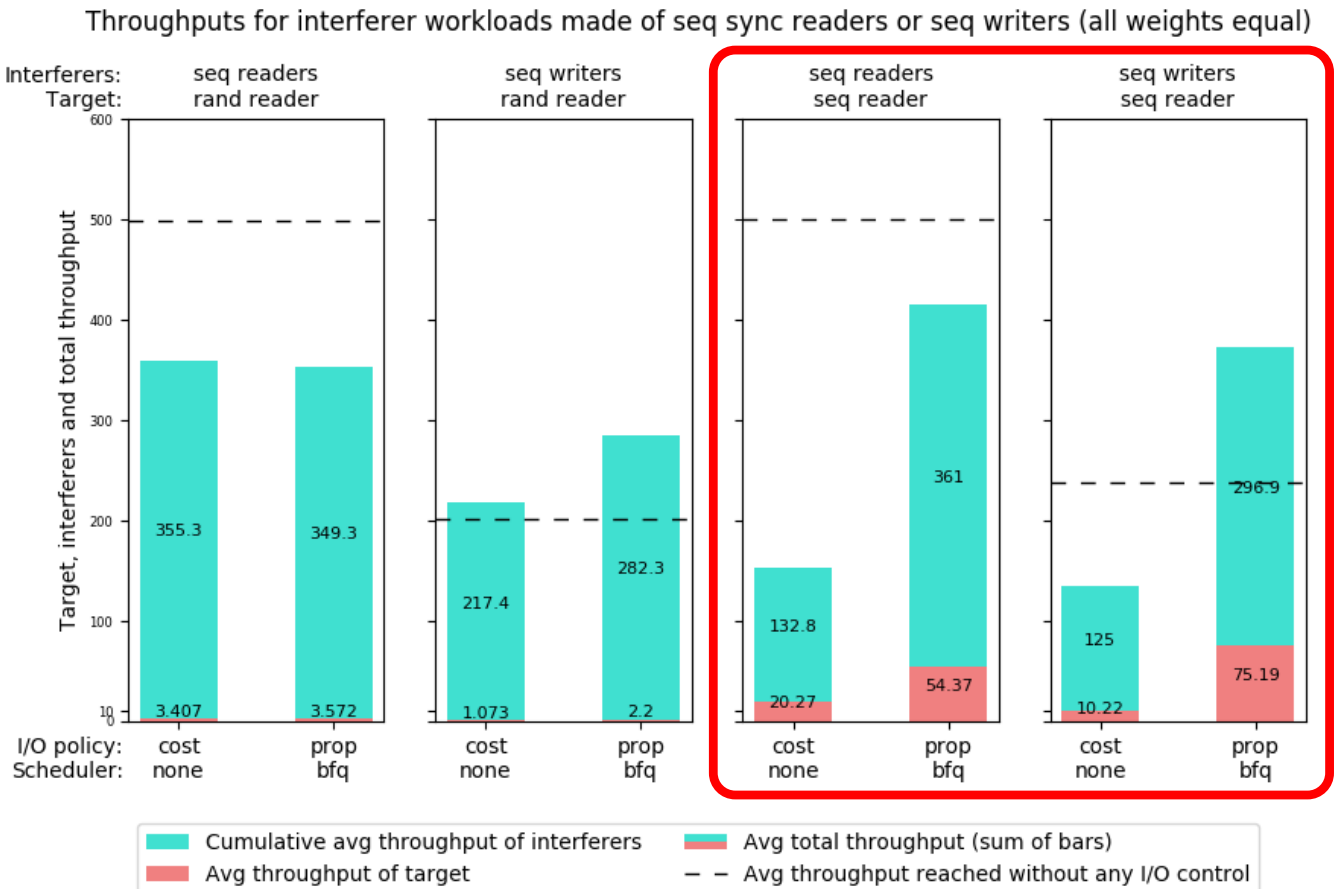
Reason for *BFQ* outperformance: failure of controllers

- For some common workloads, the new controllers failed to
  - Provide the expected latency or bandwidth to groups
  - Reach a high throughput
- That comparison however did not answer the following, important question
  - Why did controllers fail?
- Answering this question is the main goal of this presentation
- For brevity, we focus only on *io.cost*
  - More complex and accurate than *io.latency*
- We consider only three of the workloads for which *io.cost* fails (on a PLEXTOR PX-256M5S SATA SSD, *ext4*)
- Full description of the plots in the above-mentioned presentation:
  `http://connect.linaro.org/resources/san19/san19-114`

Example of failure in guaranteeing bandwidth



Examples of failure in guaranteeing throughput



Tracing *io.cost* internals

- To analyze the cause of these failures, we made a patch to trace internal parameters: `https://algo.ing.unimo.it/people/paolo/BFQ/iocost`
- Traces comply with the following cause for these failure

Root cause for all failures

Drives have very complex transfer functions

- Because of multiple channels, in-channel pipelines, striping, locality-dependent parallelism or cross-workloads interference, *readahead*, I/O-request reordering, garbage collection, wearing, ...
- Parameters of transfer functions are
  - non-linear
  - time- and workload-dependent
  - hard to know or compute precisely

io.cost (and *io.latency*) control I/O through these parameters

- *io.cost* uses I/O costs and device saturation to control I/O

I/O-cost imprecision and weight feedback loop

- *io.cost* measures service received by each group as a function of I/O costs
- Modeled I/O costs depend only on I/O type and size
- But actual I/O costs **vary with time and I/O patterns**
  - Even as a function of how *io.cost* itself modifies the I/O pattern by throttling groups
- The resulting deviation between modeled and actual I/O costs causes *io.cost* to misestimate the service received by groups
- To address this issue (only in terms of consequences on total throughput), *io.cost* does modify weights dynamically
  - This feddback loop fails too, as we show in extra slides

Device saturation, *vrate* and busy level

- *io.cost* dispatches I/O at an overall rate proportional to *vrate*
  - dynamically adjusted so as to keep drive always busy, but not overloaded
  - computed as a function of *busy level*
- Busy level evaluated as a function of
  - Numbers of groups in service *surplus* or *lagging* behind target service
  - I/O-request latency
    - Compared with static, user-modifiable thresholds
- Imprecise busy-level evaluation
  - surpluses and lags affected by I/O-cost imprecision
  - relation between latency and actual device saturation depends on I/O patterns
- We can finally see the consequences of all these issues combined, in the throughput failure with only reads

Consequence

- Depending on the actual transfer function, it may be hard for *io.cost* to control per-group bandwidth and boost throughput stably and effectively

Real-life example

- Got lost with so many details?



- A real-life example may help grasp the essence of the problem
- Then we will apply the previous facts to *io.cost* failures

A real-life feedback-control example

- Consider a building where little or no care has been put in implementing a stable and easy-to-control water-heating system
- Suppose you want to take a shower in such a building
- Getting the shower temperature right may be a problem
- Knob rotations stimulate, non-linearly, a non-linear system that reacts with time-varying delays

Ouch!

- You may know its parameters too little to successfully control the temperature!



And ... damn!

- Even after you make it, external events may change parameters so much to make you burn ourselves!

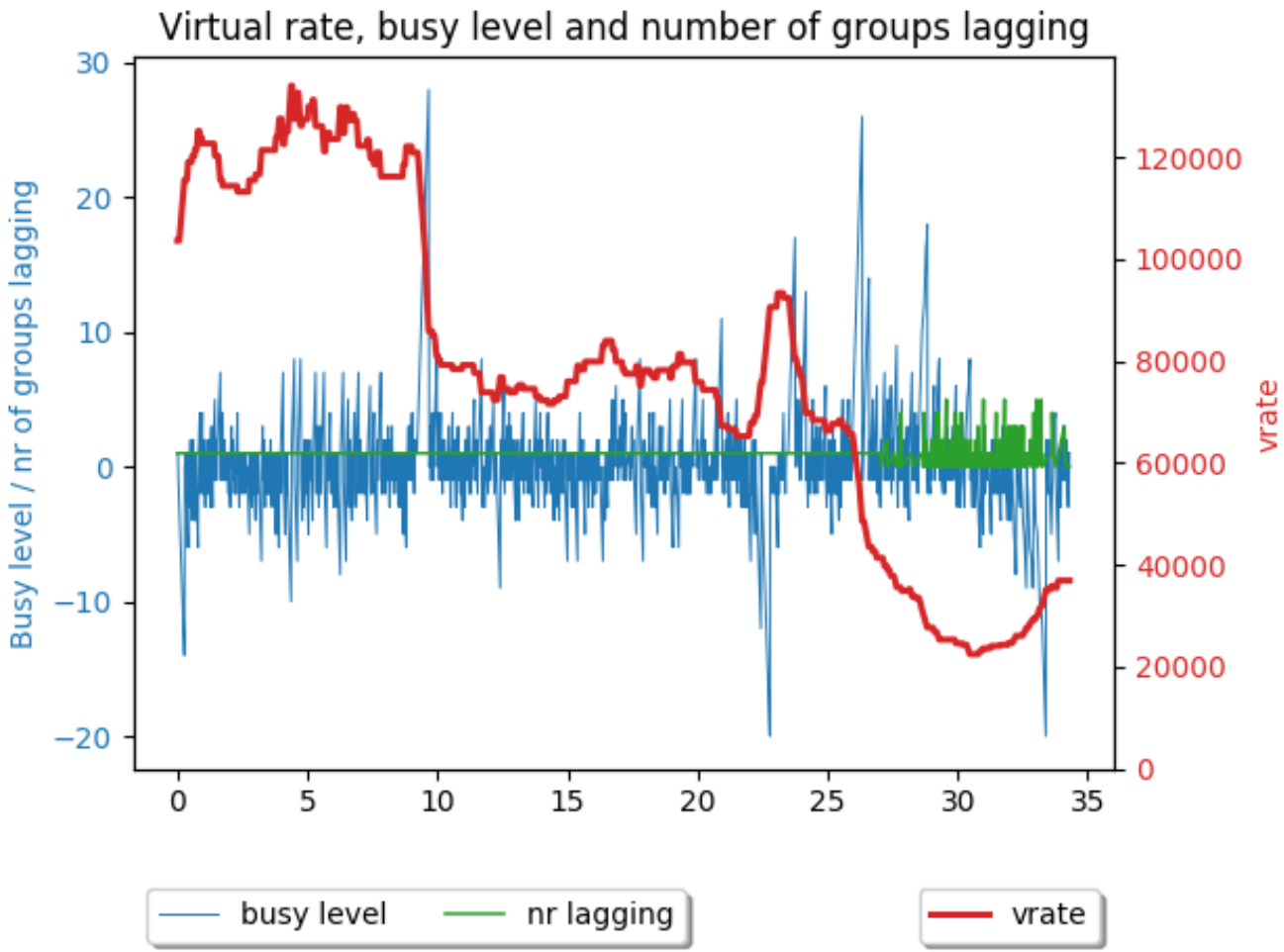


Back to *io.cost*: failure to reach a high throughput

- For *io.cost*, guaranteeing fairness while reaching a high I/O throughput is as difficult as getting the shower temperature right in such a building
- For brevity we focus only on throughput failure
- Extra slides available about the other failure

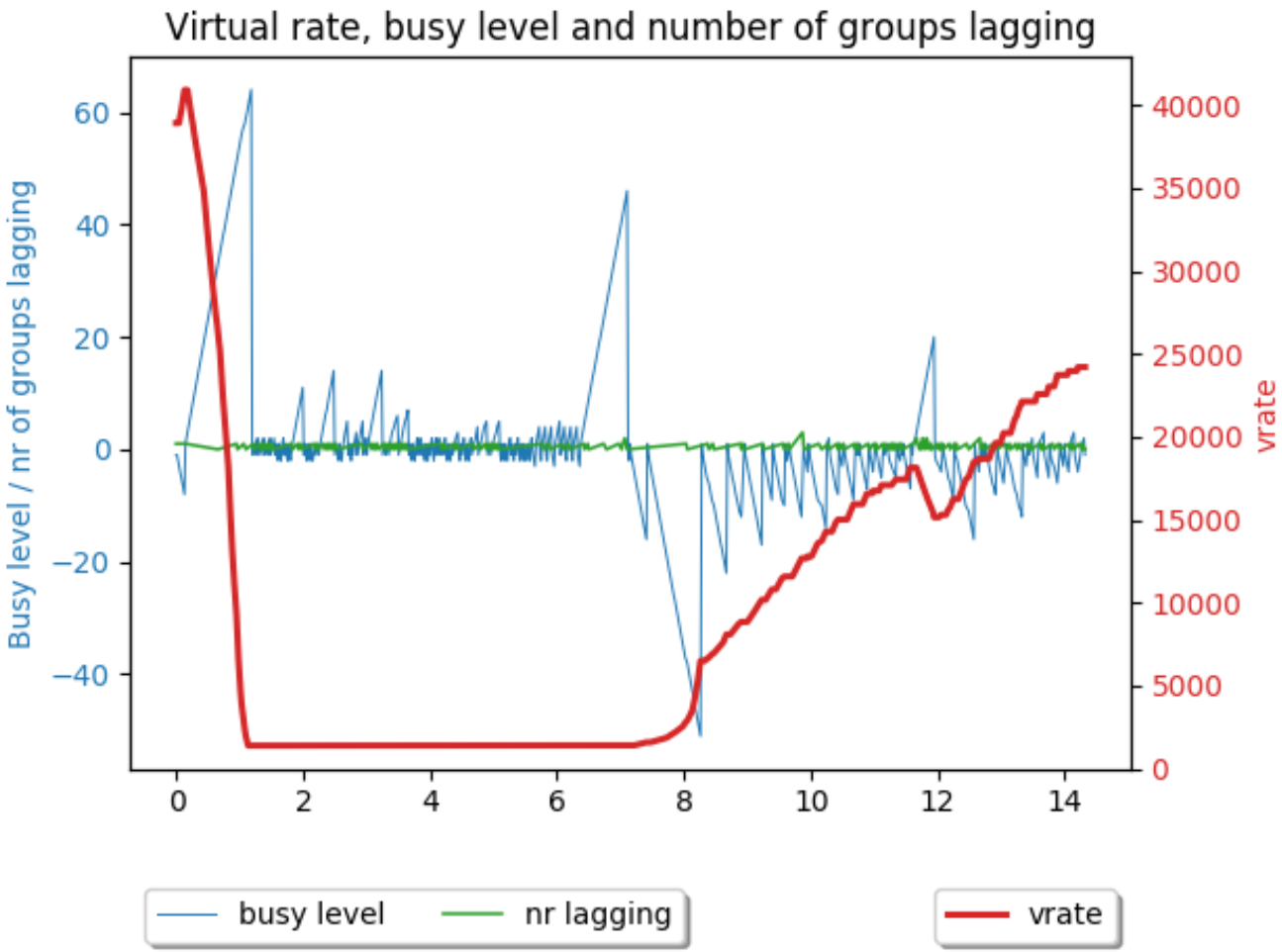
False saturation detections during throughput failure

- Recurrent false detections of high busy levels cause *vrate* drops
- Presence of groups lagging makes rise of *vrate* slower



Same problem in the failure with writes

- Dramatic false detections of device saturation, followed by late and slow recovery



Wrapping up

- *io.cost* and *io.latency* certainly skillfully crafted and tuned to get the shower temperature right on the authors' machines
- But if transfer functions vary, it may be hard or impossible for *io.cost* or *io.latency* to control I/O with some workloads

That's all folks!



Questions?

Extra slides

- Next slides address the failure in distributing bandwidth

I/O-cost imprecision and weight feedback loop

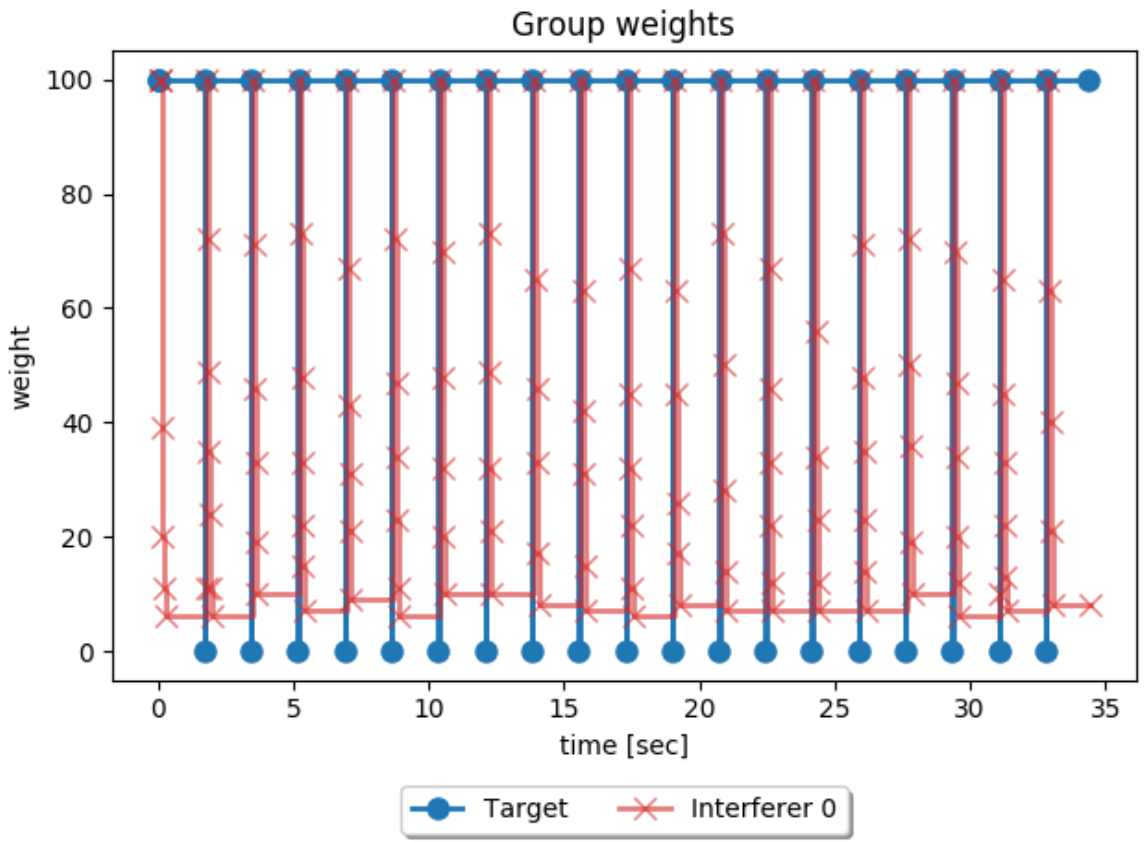
- The deviation between modeled and actual I/O costs causes *io.cost* to misestimate the service received by groups
- *io.cost* does take these errors into account, only to recover their consequences of total throughput
  - Modifies weights to counter the loss of throughput caused by wrong estimates
  - But with a simple, heuristic feedback control loop that does not take into account how and with what delay the control loop itself modifies both I/O costs and any other relevant parameter

Tracing group weights in *io.cost*

- With our instrumentation patch, we traced the values of the weights shown in next figure
  - Only for the target and one interferer
  - The same happens for all interferers

Group weights during fairness failure

- *io.cost* cyclically thinks the interferer is getting too much service, and reduces its weight. Then it realizes that the interferers is lagging, and resets weights



Fairness failure explanation

- Essentially because of wrong I/O costs, *io.cost* throttles interferers too much, and lowers interferer weights too much