SFQ(D^2): Start Time Fair Queueing with adaptive, dynamic Depth

Douglas Otstott *Wenji Li*

1 Introduction and Background

Recently, fair queuing schedulers, originally intended for network packet switching, have been adopted as I/O schedulers. Specifically, SFQ [1] and SFQ(D)'s [3] scheduling logic have been re-purposed for simple host-side I/O devices. One notable example, FlashFQ [4] attempted to design a flash-optimized fair IO scheduler, using a slightly modified version of the SFQ(D) algorithm.

The authors of FlashFQ argue that SFQ(D) is especially useful for fair scheduling multiple processes on flash due to the highly parallel nature of flash devices. Since a single flash device may consist of an array of flash die which are capable of independent operation, the author's adopted SFQ(D)'s concept of "depth" in which they profile devices offline to determine some optimal depth value, treated as input D. They also claim fair queuing is superior to time slices, as they eliminate periods of process unresponsiveness on parallel devices.

While FlashFQ was able to achieve significant improvements in I/O performance, the authors were forced to implement some detrimental mechanisms in order to achieve adequate fairness. Specifically, FlashFQ implements anticipatory scheduling [2] in order to overcome deceptiveness idleness, a well known concept and common approach universal to anticipatory schedulers. Basically, the scheduler halts operation for some small window (2 ms) anytime a process completes all its requests. The argument is: a process performing synchronous I/O may, unfairly, forfeit its allotted time slice if it appears to go idle between requests. The authors claim by waiting they prevent such processes from losing their share of the device in this situation. They support this claim with evaluation results. Furthermore, FlashFQ throttles specific processes if they are 'progressing' significantly faster than other processes. The authors define progress as the start value of process' last issued request. This situation can arise if one process is issuing sustained large, intensive requests (i.e. writing a log) while another is issuing sparse, non-intensive requests (i.e. synchronous reads). FlashFQ solves this issue by blocking requests from the faster process until the slower process can catch up, a mechanism called *Throttled Dispatch*.

Both of the aforementioned polices(anticipation and throttled dispatch), are useful for enforcing fairness guarantees. However, they fail to maintain the principle of work conservation: they stall the device while requests are queued in the scheduler. The benefits of anticipatory scheduling are well studied in traditional hard disks. However, the benefits of anticipation are due, largely, to the hardware constraints of the device: hard disks exhibit no parallelism and most of their latency is determined by the disk arm's seek time. Flash devices, however, lack an arm, do not have a seek latency and are highly parallelized. It is for these reasons, we doubt the contribution of anticipation (and similarly, dispatch throttling) in flash I/O schedulers.

Additionally, simple request depth may not be enough to capture the optimal parallelism of a flash device. The authors of FlashFQ profiled their devices via offline benchmarking in order to determine the optimal number of outstanding requests. However, based on known issues with flash performance, we believe the optimal depth of any given flash device may be variable. Internal states set by garbage collection, data fragmentation and device utilization may result in a different optimal depth at different times.

In this paper, we propose SFQ(D^2): a start-time fair queuing scheduler with a dynamic depth value. We assert that the purpose of profiling should be to determine the optimal performance of a flash device and the schedulers job should be to vary the IO depth based on discrepancies between observed performance and target performance. SFQ(D^2) will be completely work conserving: the scheduler will never let the device go idle while there are requests in the queue. Instead, SFQ(D^2) will replace FlashFQ's anticipation and dispatch throttling with adjustments to the IO depth.

2 Project Goals

The goal of this project is to implement a start-time fair queuing IO scheduler for flash devices. This scheduler would have a variable I/O depth attribute(D), and dynamically calibrate D in order to achieve and maintain target performance parameters. This scheduler would be work conserving (so D will always be greater than 1), but the scheduler should strive to maintain inter-process fairness in terms of device usage so it may shrink D in order to do so. The name of the proposed scheduler is SFQ(D^2) or Start-time Fair Queuing with Dynamic Depth.

2.1 Design

Target performance will be characterized by per request latency and overall device bandwidth. Fairness with be determined by per process relative progress. Relative progress is determined by the difference between the start times of the last issued request from two processes.

SFQ(D^2) will optimize performance while guaranteeing fairness by tuning depth using a control loop. This loop will maintain moving average data about request latency and bandwidth for the last several requests as well as minimum and maximum observed values. It will also consider the relative progress of I/O issuing processes. Part of this project will be to accurately model the relationship between latency, bandwidth, depth and fairness and implement it in this control loop. On the surface, latency and fairness should benefit from reduced depth at the cost of bandwidth and vice versa, but more work is necessary for the final paper.

2.2 Evaluation

For the final paper, we intend in implement SFQ, SFQ(D), FlashFQ, and SFQ(D^2) as I/O schedulers on a real Linux system and evaluate them with a real flash device. Evaluation data would entail performance benchmarking with per process responsiveness and latency statistics as well as global device bandwidth. We believe that due to the work conserving behavior of our scheduler, we will outperform the aforementioned schedulers in device performance while matching inter-process relative responsiveness.

3 Current Status

Since we spent a lot of time discussing the feasibility of this project, we only recently finished SFQ(D^2)'s design and began implementation.

Once we fully understood the SFQ and FlashFQ papers, we listed the points that we could improve. Our next step is to implement SFQ in Linux. A breakdown of future work is summarized in Table 1.

Date	Works
Feb 17 - Feb 28	Finish the design for SFQ(D^2)
Mar 1 - Mar 15	Basic SFQ Implementation
Mar 16 - Mar 31	SFQ(D) Implementation
Apr 1 - Apr 15	FlashFQ Implementation
Apr 16 - Apr 30	SFQ(D^2) Implementation
May 1 - May 15	Evaluation on SFQ SFQ(D^2) and FlashFQ

Table 1: Bi-weekly schedule

4 Deliverables

By the end of the project, we plan to implement SFQ, SFQ(D), FlashFQ and SFQ(Q^2) as IO schedulers in Linux. And the expected evaluation result will show that the SFQ(D^2) can utilize the parallelism property by tuning IO depth better than FlashFQ, maximize performance and guarantee fairness.

References

[1] GOYAL, P., VIN, H. M., AND CHEN, H. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. *SIGCOMM Comput. Commun. Rev. 26*, 4 (Aug. 1996), 157–168.

[2] IYER, S., AND DRUSCHEL, P. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), SOSP '01, ACM, pp. 117–130.

[3] JIN, W., CHASE, J. S., AND KAUR, J. Interposed proportional sharing for a storage service utility. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2004), SIGMETRICS '04/Performance '04, ACM, pp. 37–48.

[4] SHEN, K., AND PARK, S. Flashfq: A fair queueing i/o scheduler for flash-based ssds. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 67–78.