I/O Scheduling in FreeBSD's CAM Subsystem

M. Warner Losh
Netflix, Inc.
Los Gatos, California
USA
Email: wlosh@netflix.com

***Abstract*—FreeBSD's default I/O scheduling policy strives for a general purpose performance across a wide range of applications and media types. Its scheduler does not allow some I/O requests to be given priority over others. It makes no allowances for write amplification in flash devices. GEOM level scheduling is available, but it is limited in what it provides. It is too high in the stack to allow precise control over I/O access patterns. A scheduler in Common Access Method (CAM) can exploit characteristics of the device to precisely control I/O patterns to improve performance. At Netflix, we found that limiting write requests to the drive limited the effects of write amplification on read latency. In addition, we found that favoring reads over writes improves average read latency, but does not fix extreme outliers.**

Updates to the software since this paper was written can be found in an appendix. The changes in code and evolution of the understanding of what conclusions can be drawn haven't been integrated through the whole paper as yet.

I. Introduction

The I/O subsystem can be thought of as a stack. At the top of the stack are user requests. At the bottom of the stack are media devices or remote connections to media devices that service user requests. Each layer in the stack may generate its own requests, combine multiple requests together, generate multiple requests for a single higher-level request, or direct requests to multiple places. Many layers may have hooks into the VM system or cache information to optimize performance.

No matter where I/O scheduling happens, it may:

- reorder or delay dispatch of I/O requests
- happen at any layer of the I/O stack
- unfairly allocate I/O resources among users
- route related traffic to the same CPU
- trade performance of one class of I/O for another

While I/O scheduling in FreeBSD is generally good, as Fig. 1 shows, some workloads cause it to have horrible read latency. The dotted red line shows normal latency of 5ms. The dashed red line shows the highest acceptable latency of 35ms. These spikes can be 100x or more above normal values. As we shall see however, the fault lies not only with the FreeBSD I/O scheduler but also with other factors.

II. FreeBSD's I/O Stack

The FreeBSD I/O stack is similar to many others. Requests enter the stack at the top via system calls. These are filtered down either to the raw disk driver, or via a file system to the disk. The layer just above the GEOM layer can be thought of as the upper half of the stack. The upper half of the stack has

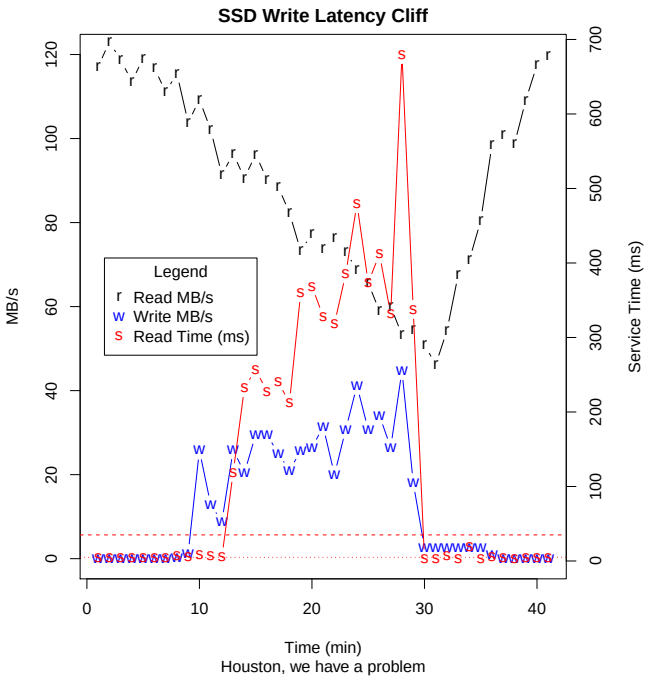

Fig. 1. Default scheduler problem workload: latency spike in red.

its own optimizations, data flows, scheduling, etc. [1] which are beyond the scope of this work. Fig. 2 shows these layers as well as the lower half.

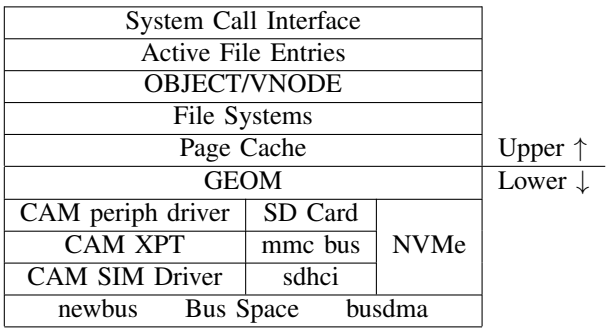

Fig. 2. Simplified FreeBSD I/O stack. After [1, Fig. 7.1]

The disk driver layer in FreeBSD is called GEOM [2] [3]. It handles the disk partitioning in a general way, and implements advanced features, such as software RAID, encryption,

compression and scheduling. Below the GEOM layer, most block devices use the CAM subsystem to manage access to the media [4] [5] [6, §14]. CAM supports a variety of peripherals (periph), transports (xpt), and SIMs.[1] CAM implements a generalized queuing and error recovery model. Although the original CAM standard [7] was for SCSI only, FreeBSD has extended it to support additional transports and protocols (USB, Firewire, Fibre Channel, ATA, SATA, and iSCSI) and to integrate SMP locking. While the vast majority of storage devices in FreeBSD use CAM, some bypass CAM and use their own queuing and recovery mechanisms. Disk drivers that bypass CAM are beyond the scope of this work.[2]

From the GEOM layer on down can be thought of as the lower half of the I/O stack in FreeBSD. The upper layers transform all their requests from `struct buf` or system calls into a common `struct bio` to pass into GEOM [8]. Requests remain `struct bio` through GEOM. Most requests that enter the GEOM layer are I/O requests that need to access the media to satisfy them. They are eventually passed to the `struct disk` drivers below.

The default I/O scheduling policy in GEOM is a simple FIFO. As the requests come in, they are dispatched to the lower layers without further buffering, ordering, delays, or aggregation. Below the GEOM layer, each disk driver sets its own scheduling policy. CAM provides two periph drivers for all disk I/O in the system: one for SCSI-based disks and one for ATA-based disks. Both drivers provide the same scheduling policy. If a BIO_DELETE is available, and none are active, it combines as many BIO_DELETE requests as it can into one command and schedules it. Otherwise, it will schedule as many BIO_READ, BIO_WRITE, or BIO_FLUSH commands with the underlying media as will fit into its command queue. The SIM drivers for SCSI requests pass the requests to the drives. The SIM driver for ATA requests (ahci) is more complicated. NCQ[3] commands are sent to the drive if the queue is not frozen. For non-NCQ commands, it freezes the queue, drains the current requests, sends the non-NCQ command then unfreezes the queue. This makes non-NCQ commands relatively expensive and disruptive. If they run a long time, they add significant queuing latency to NCQ requests that arrive during execution.

These general purpose algorithms work well enough for most users of the system. Most access patterns fall neatly into one of these two patterns. FreeBSD does offer a GEOM module that can schedule I/O and associate I/O with different queues tied to individual cores [9]. This scheduling algorithm works well for a very narrow workload where multiple high volume streams. The code in `geom_sched` provides a good framework, but the default `rr` scheduling algorithm is highly tuned for devices that match a 7200 RPM SATA hard drive.

Netflix found FreeBSD's scheduling of I/Os to exhibit the following undesirable traits:

- uneven latency for I/O in the system with extremely long read-latency tails (see Fig. 1)
- no provisions to trade write performance for read latency, or vice versa.
- no way to limit the effects of write amplification in SSDs when they fall behind and the extra writes reduce performance
- no way to bias read requests over other requests
- no way to limit the elevator's car size in `bioq_disksort`
- no deadline to submit I/O to drive, which can lead to long latencies for spinning disks.[4]

III. Flash Drive Characteristics

Newer SSDs and other NAND flash devices offer a quantum leap over traditional spinning disks. Devices built on NAND flash reflect the NAND's underlying characteristics [10] [11]. Conceptually, SSDs and NVMe cards can be modeled as

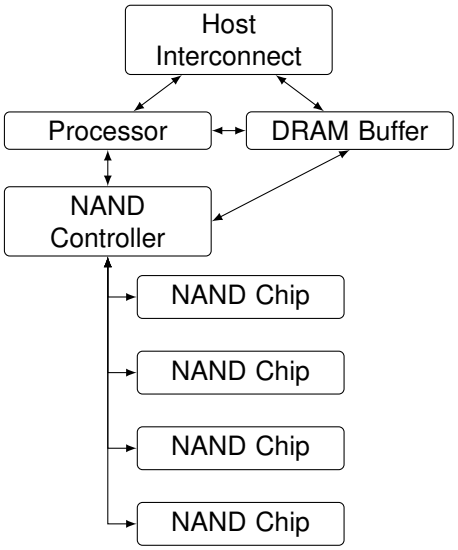

Fig. 3. Generalized block diagram of flash storage device.

shown in Fig. 3. Drives connect over some transport layer (SATA, SAS, PCIe, etc) via the Host Interconnect and present a continuous range of blocks that can be read, written, or trimmed using Logical Block Addresses (LBAs).[5] The processor on the drive converts LBAs in the host commands to physical addresses on the NAND flash. The Flash Translation Layer (FTL) runs on the processor and handles the translation between the LBA and the physical NAND location, free pool

[1]Originally an abbreviation for "SCSI Interface Modules," a SIM can now be an interface module to any transport protocol. It is best to think of SIM as a driver for any HBA (Host Bus Adapter).

[2]Generally these are drivers written before CAM became transport agnostic: SD cards, raid controllers, really old CDROM drivers. However NVMe bypasses CAM and is quite recent.

[3]NCQ — Native Command Queuing allows up to 32 tagged commands to be sent to the drive at one time; however NCQ and non-NCQ commands cannot be mixed. To execute a non-NCQ command, all NCQ commands must finish first.

[4]Improving the read latencies on spinning media is beyond the scope of this work.

[5]Numbered from 0 to N, contrasted with old addresses that specified cylinder, head, and sector.

management, and wear leveling. The DRAM buffers the I/O requests, drive responses, and the data streamed between the host and flash. The NAND controller handles addressing the NAND, sending it commands and streaming the data to and from the DRAM. The processor and NAND controller work together to cope with the difficult quirks NAND chips have to hide them from the end user.

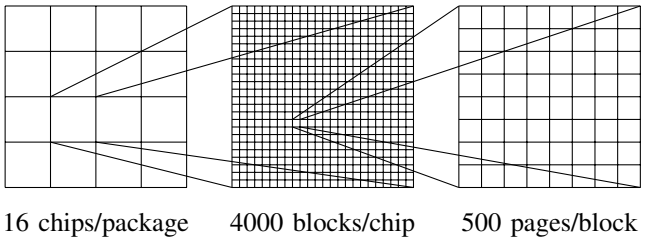

Fig. 4. NAND typical geometry.

As illustrated in Fig. 4, NAND flash is organized hierarchically. Each flash package contains a number of die (typically a power of 2 up to 16). Each die has a number of erase blocks (typically a few thousand), which is the smallest unit that can be erased. Each block contains a number of pages (typically a few hundred). Each page is a power of two bytes, usually between 4k and 32k, plus a few percent extra for ECC protection and metadata. The page is the smallest unit that can be read or written.[6] Pages can only be written once without being erased. Some technologies impose additional requirements. MLC NAND[7] generally must be written sequentially. Different chip families may have additional requirements dictated by their geometry. For example, for long-term data integrity, pages within a block may need to be written as quickly as possible. Some designs can write to a few blocks at a time to facilitate garbage collection [10], reducing garbage collection's effect on user I/O performance. Some designs have special data movement commands for garbage collection as well.

The relationship between large erase blocks and smaller pages (the typical I/O size) forces a log-structure for nearly all devices [12]. A log is a sequentially written data structure. A log-structured device appends data (and sometimes metadata) blocks to the end of a log for writes. Reads are translated from an LBA to the physical address. As LBAs are rewritten or trimmed, holes develop within the log. When new blocks are needed for data, the firmware moves old data from older blocks to the end of the log and then erases the old block. After this movement, the LBAs in the blocks are fully compacted, freeing up the holes from the old blocks. The firmware keeps metadata about these translations, which pages are used within blocks, and other miscellaneous statistics [13].

The amount of garbage collection and data movement directly affects performance. To find space for new data to be written to the drive, its firmware must garbage collects empty or mostly empty blocks. This garbage collection generates additional read and write traffic for each write issued to the drive.

Definition 1 (Write Amplification): In a log-structured system, write amplification, A, due to garbage collection is defined as the average of actual number of page writes per user page write:

$$A = \frac{U + S}{U}$$

where U is the number of user writes and S is the number of additional writes generated by the drive firmware for garbage collection or other reasons. The closely related *Write Amplification Factor* can be expressed as:

$$A_f = \frac{S}{U}$$

implying that for each user write, $1 + A_f$ writes are written to the underlying media [13].

Write amplification is dependent on the workload, drive history, and design choices of the firmware implementation. In most typical workloads, write amplification varies over time but tends to settle around some average. Some drives provide a feedback mechanism to discover the current A or A_f. However, this information is either vendor specific, unavailable, or too costly to retrieve. These limitations make it hard to use A or A_f in an I/O scheduler.

To help mitigate these issues, drive vendors have defined commands to inform the drive when data is no longer needed. In SATA, these commands are called Data Set Management (or sometimes just TRIM, after the bit in the DSM command) [14, §7.5]. SCSI defines different data deletion methods (UNMAP is the term used) [15, §7.4]. Although the exact details differ as to persistence, reliability, and predictability of the data after a TRIM, all variations inform the drive of LBAs no longer in use. TRIMs tell the drive when blocks no longer contain useful data and can be freed. This increases the free pool of blocks and reduces the data that must be copied forward. In theory, this should reduce the effects of write amplification as the S term will be reduced. In practice, the time it takes to execute the TRIM command may negate this savings.

One other consideration is efficiency. The relationship between the TRIM size and time to execute is often non-linear. The exact type of TRIM to use can be important as well, since some TRIM commands can be queued and mixed with other I/O, while others force all other I/O to drain before the command can be issued and block new I/O from being sent to the drive until the command completes. This latter form can have a large effect on read latency for those reads forced to wait for a TRIM to complete. Unfortunately, the ATA disk driver in CAM uses only non-NCQ TRIM commands, which increases the latency of all I/O. Other SIMs talk to devices that do not have the interface to set the registers needed to execute an NCQ DSM TRIM. Some controllers do all the queue management for requests to the drive. This makes I/O going through these SIM drivers more unpredictable because control over each individual transaction's timing is up to the hardware, not the host.

[6]Large-page designs have sub-page reads to increase small request IOPS.
[7]Multi Level Cells, which encode two or more bits per cell.

Many have observed a large variation in quality of implementation between different drive models and manufacturers. This evaluation of a new SSD [16] shows data from a range of SSD models. The results varying by an order of magnitude between models depending on the benchmark. These differences can lead to substantial performance variation for certain workloads. For example, during development of this scheduler, a certain manufacturer's drives showed large spikes in latency when the write traffic to the drive was above a few percent of the spec sheet value. When the drive had 98% reads and 2% writes (approx 150MB/s read, 5MB/s write), the latency for reads was characteristically 2–3ms. However, when this changes to 90% reads and 10% writes (approximately 150MB/s read, 15MB/s write), read latency would spike to into the hundreds of milliseconds. Investigation of traces showed that some of this was due to some long writes (50–100ms) and some long, blocking TRIMs (up to 200ms). The behavior was not consistent because short bursts of writes did not exhibit this degradation, and some TRIMs returned quickly (10ms).

IV. Netflix's Workload

According to published reports, Netflix accounts for a little more than one-third of the peak download traffic in the US [17] [18] [19]. Netflix embeds its Open Connect Appliance (OCA) [20] at various locations in the Internet to stream video to its customers. There are two main types of OCA appliance: storage and flash. The storage appliance stores the entire catalog on spinning disks but can serve only a limited number of clients. The flash appliance stores the most popular content on SSDs and can serve many more clients but has a smaller storage footprint. During off-peak times, these boxes update their content on a staggered schedule. New content is licensed, old content expires, and content is occasionally re-encoded to fix problems or increase efficiency. In addition, popularity changes over time. During peak hours, no changes are made to the box since at peak load there are not enough resources in the box for updates. Averaged over a day, OCAs read between 1,000 and 10,000 times more data than they write.

Conventional wisdom states video streaming workloads are sequential [21]. However, I/O traces of the OCA at the disk interface show it to be a random workload. Although each video is sequential on the disk, two factors confound the sequential pattern. First, when multiple clients are streaming the same title, at the same bit-rate, they rarely all start at the same time. This temporal displacement of clients effectively makes each one an independent stream. Second, the sheer number of clients (30,000–40,000 on a busy flash OCA) randomizes things even more.

Netflix has established metrics to gauge the quality of the customer experience with its streaming service. These metrics include a measure of read latency for the media in our systems. When latency gets too high or variable, customer experience suffers as the factors necessary for smooth, high-quality playback are not all present. To keep playback quality high, and to avoid serving when latency is likely to be high, OCA servers take a break from serving clients while refreshing their content. This fill window is a few hours during the off hours.

A frequent request from our network operations team for OCAs is to allow filling while content is being served during off-peak hours. Operationally, when a server is filling its new content, it is not serving clients. This takes capacity out of the network during off-peak hours. Our flash systems do not serve traffic while filling due to the nature of the SSD drives that are in them. When even a small stream of writes is going to the drive, its read performance plummets, leading to latency spikes large enough to disrupt streaming to clients.

FreeBSD's default I/O scheduler treats all I/O the same and schedules it in FIFO order to flash devices. This in-order scheduling is well suited for a general purpose operating system. However, there is no way to limit the writes to a drive to lessen the effects of write amplification. Unlike network interfaces, there is no way to do any kind of traffic shaping to the drive. There is no way to prioritize your most important traffic. There is no way to monitor the time that I/Os are in the media's hardware buffers before being satisfied. While the default I/O scheduler does a fair job with these things (good enough for Netflix to serve the traffic in the previously cited reports), it lacks the fidelity of choice necessary to intelligently shape traffic to the drive to keep it operating within its optimal envelope.

Fig. 1 illustrates a specific problem we observed in our systems on many occasions. This data is from one of our flash cache machines when a fill of content happened while serving video to clients (overriding the defaults to not do this). We see a read rate of about 120MB/s with a latency of about 4ms. This is about 40% of the drive's data-sheet capacity of 300MB/s read, 400MB/s write. About nine minutes into this data set, the application starts writing at about 20MB/s, giving about an 85%/15% read/write ratio. four minutes later, read latency spikes to hundreds of ms. Our system control infrastructure notices the high latency and reduces traffic to this node reducing SSD reads. After ten more minutes, update completes, the write load ceases, and read latency returns to normal. Our control infrastructure notices and starts sending more traffic to the box, restoring the 120MB/s read rate. These data illustrate the problem, though the exact numbers vary from run to run. We created the Netflix I/O scheduler to address this, and similar, problems.

V. FreeBSD I/O Scheduler

Fig. 5 shows the flow of `struct bio` I/O requests down from the upper layers through GEOM and into CAM. In this diagram, `simaction` represents the `action` routine passed to `cam_sim_alloc` when the SIM was registered. `sim_intr` is the SIM's interrupt routine registered with newbus. The diagram shows the flow for a SCSI direct access device (da driver).[8] The flow for the `ada` driver that handles

[8]By convention, all routines from the `da` driver start with `da`, eg `dadone`, `dastart`, etc. Similarly for the `ada`. Sadly there is no `dabulls` routine, much to the chagrin of Chicago Bulls fans [22].

SATA devices is the same.

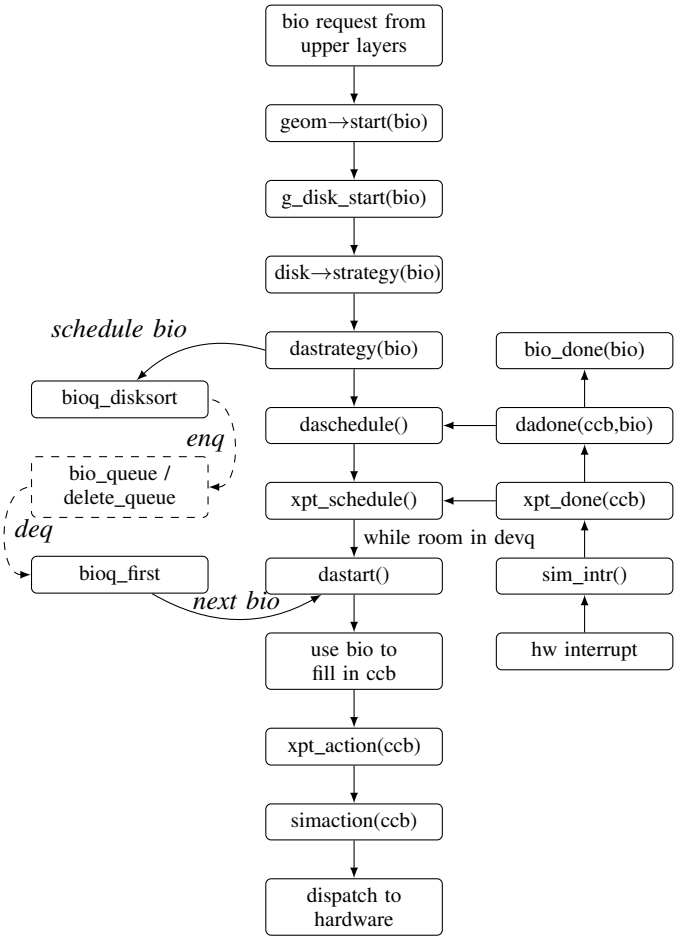

Fig. 5. Disk I/O data path through CAM [23].

GEOM transforms the `bio` into the proper absolute offsets on the disk based on the relative offset in the partition. It passes this to the disk's strategy routine. The `da` periph driver places the `bio` onto a queue and calls `daschedule` to tell CAM to allocate a `struct ccb` (CAM Control Block) if any slots are available in the periph's `devq`. CAM keeps track of how many requests are outstanding to a periph in its `devq` and calls `dastart` only if there are slots free (otherwise it will just return). Once `dastart` is called, it will try to pull a request from the `bio_queue` and use a request to fill in the `ccb` with the appropriate command block for the request. `xpt_action` puts the `ccb` into the SIM's queue and call its `action` routine. This delivers the command to the hardware. Later, an interrupt will fire, indicating the command has completed and call the SIM's `poll` routine. The SIM will then call `xpt_done`, which calls `xpt_schedule` if there is a free slot in the device's `devq`. `xpt_done` then call the request's done routine `dadone` which completes the `bio` request by looking at the status in the `ccb` and calling either `bio_done` when there is no errors, or `bio_error` to signal an error. The order here is important, and different than typical.

The current I/O scheduler is so simple that you might have missed it. In Fig. 5, it is the boxes in the left-hand column. Today, the disk's strategy routine simply places the I/O onto the `bio_queue` using the elevator sort algorithm discussed earlier (for flash devices, it simply places it at the end of the queue since the elevator optimization provides no benefit and has the in-order insertion cost associated with it). The start routine then pulls the first I/O off the queue and sends it to the device. When the I/O completes, the ISR signals the completion up the stack. BIO_DELETE requests are special, as explained above, since only one TRIM command can be active at a time.

VI. Netflix Scheduler Theory

We know from basic design theory of NAND-based flash storage that write amplification is a concern [13]. Write amplification is the biggest concern when you invalidate part of the drive and write new LBAs (either from the range invalidated or not). Write amplification has been observed in our systems to be 3 or 4 at times (though the long term average is between 1 and 2). Some drives, which do not support querying write amplification, exhibit behavior that suggests an even higher write amplification due to max write throughput being less than 5% of the data-sheet write rates. Furthermore, we know from the chips used in our drives that the controller has a limited number of concurrent channels available for reading and writing. During garbage collection, blocks must be erased before their pages can be written to.

A relatively low write amplification can cause problems. If you allow up to 32 writes to the drive at one time, with a moderate I/O size, you can easily consume a significant portion of a logical NAND block with in-flight data. To satisfy those writes, the controller must read and write several blocks when space is low and it has no free blocks to allocate to the I/O. This causes several banks of the system to be used at once. To write a block, you have to erase it first, which is a very time consuming operation for NAND today. The more banks that are involved in writing data, the fewer that are available for reading. NAND chips can process only one request at a time, be it erase, read, or write. If a bank is writing data, read requests wait for that to finish. Pending reads queue up behind this activity, leading to a fairly long time to service the read, as well as long wait times to dispatch the reads to the drive.

The Netflix scheduler implements a two-pronged attack. First, priority is given to reads over writes when the scheduler must pick which one to send to the drive. For Netflix's workload, reads are the most important thing, and we are willing to trade write performance to get extra and more sustained read performance. Second, to limit the number of banks that will be affected by write amplification, the scheduler limits the number of outstanding write requests that are sent to the drive. With fewer write requests in the queue at any time, the drive is necessarily limited to one bank for most likely garbage collection scenarios. By limiting the in-flight writes to a small fraction of a logical NAND block, and knowing that our drive vendors over provision the NAND chips by

about 10% according to the data sheets, we hope to limit data movement from garbage collection to less than one logical NAND block of data. With this limit, at most two banks of the drive will be busy. It appears that our vendor has 8 banks, so during the time it takes to write to the drive, three-fourths of the banks suffer no delay. One-fourth of the banks will suffer delays by contenting with garbage collection. We bound this contention, however, to approximately the erase time for a block (or about 5-10ms). Without visibility into the FTL, the I/O scheduler is unable to avoid sending down reads that would conflict (tying up queue slots that could otherwise be used for useful I/O).

VII. New I/O Scheduler Design

The design implemented gives different treatment to different classes of I/O based on configurable options. The fundamental flow is unchanged from the stock FreeBSD scheduler, with one exception. Where the current design has a single `bio_queue`, the new design splits it into separate queues for reads and writes. TRIMs retain their own queue. The new design allows the entire I/O scheduler to be swapped out, allowing for different schedulers to be used based on the needs of the system. Fig. 6 shows the new additions in red.

The new design, however, abstracts out all the scheduler decisions currently from the periph driver and puts them into a library. We modified the periph drivers to use this new library. Most scheduling decisions have been moved into this library, although some decisions related to TRIM remain in the periph drivers. Since the performance of TRIM differs between SCSI and SATA devices, it is not clear if forcing the same policy for both would be optimal.

Two schedulers were implemented for this project: control and Netflix. The control scheduler is functionally identical to the current I/O scheduler. It uses the new scheduler interface to implement the same policy as the current scheduler with one queue. This scheduler ensured identical performance and functionality when compared against the old code. When we found a difference, we fixed this scheduler to conform to the old behavior.

The Netflix scheduler we developed behaves very similarly to the default scheduler, unless any of its limiters are enabled. Two limiters were implemented: fairness bias and queue depth limits. First, we implemented a read vs. write fairness bias. We choose one type vs. the other with a predicable bias when the queue has both types in it. Second, we implemented a read, write, or TRIM limiter. When in effect, the scheduler will not queue more than the limit for each type of I/O request to the SIM, even when additional `devq` slots are available. These limiters are independent and can be enabled separately or together.

The new calls to the scheduler from the periph drivers are shown in red in Fig. 6. These routines implement the scheduling algorithms. In addition to abstracting the code into a library, we made one minor change to the periph drivers. When limiting I/O requests, we had to change the behavior of the `dastart` routine slightly. It now will sometimes return with `devq` slots available, even though the driver has requests in one of its queues. Prior to our changes, this was never the case. Since `dastart` is called by `xpt_done` before the `ccb`'s done routine, no I/O will be scheduled (since `dadone` hasn't run to update the counters for `dastart` to know it can schedule data. When completing an I/O from a limited queue, we had to add a call to `daschedule` to ensure `dastart` was called again after the limiting was no longer in effect and the counters had been updated. This change is why the arc from `dadone` to `daschedule` was made red in Fig. 6. Without this change, I/O requests could become stuck in `da`'s queues after any queue reached its limit. The added call keeps the I/O requests flowing. Finally, the `bio_queue` was split into a separate read queue and write queue to simplify the implementation of the limiters.

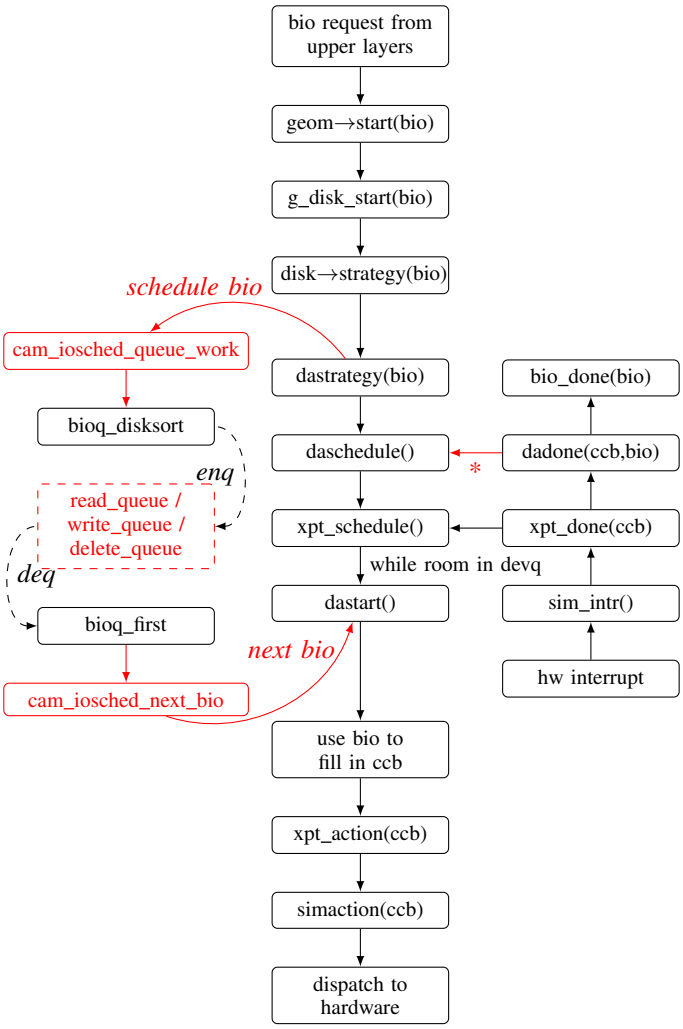

Fig. 6. Modified disk I/O data path through CAM [23].

VIII. Results

As we saw in Fig. 1, read latency was extreme in the presence of writes. To see if this resulted from some bias in the FreeBSD I/O scheduler, we tested our implementation just

turning on the read bias code. This code gave a 100 to 1 bias to read requests when a choice had to be made between reads and writes but otherwise let all requests through when there was no conflict. We had hoped this would produce useful results. We found a small (1%–2%) improvement in average latency but continued to observe the extreme read latency outliers. Seeing the outlier behavior unchanged lead us to conclude the fault was not in any bias against reads in the FreeBSD I/O scheduler.

We next wanted to test the theory that these outliers were caused by the drive needing to garbage collect. We learned from the manufacturer that the drive would automatically garbage collect blocks from TRIM requests sent down, but only after the interface had been idle 100ms. Since we were pushing thousands of IOPS through the device, the interface would never be idle that long. Modifying the scheduler to hold off read and write this long requests to allow for automatic garbage collection to take place was out of the question. This meant, unfortunately, that garbage collection happened in real time when the drive needed space for write requests. Our test data suggested that after writing about 5–10 GB we would see a big spike in read latency. This is less than 1% of the drive's capacity, but daily updates to the OCA typically were much larger. Whatever the drive was doing to keep a pool of free space available was insufficient to keep up with unlimited writes an update triggered.

To test this theory, we turned on write limiting in the Netflix I/O scheduler. We set the write queue limit to 1. This would have two effects. First, it would limit the number of concurrent writes sent to the drive, keeping in-flight data smaller than a logical NAND block. Second, it would rate-limit the writes, which would rate-limit the write amplification from garbage collection. Our tests showed that the extremely large read latency outliers had disappeared. Contrast Fig. 1 with Fig. 7, which are graphed with the same vertical (performance) scale. The graph shows that the write rate is about 25% lower than before, but the read latency no longer spikes up by a factor of 100x, but is limited to only about a 4x rise (20–30ms) in general (though sometimes 20x spikes do occur). The situation is much better but still not yet ideal. *Note:* the test that generated this data was a much larger fill designed to last for hours instead of minutes.

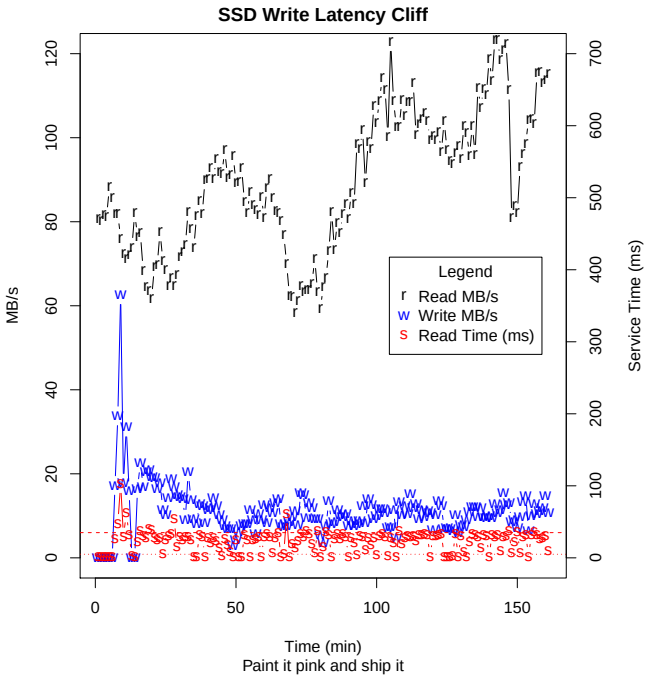

Fig. 7. Modified scheduler on problem workload.

IX. Future Work

After we implemented the low-level I/O scheduler in CAM a number of improvements suggest themselves. Some of these are quite general, while others are fairly specific.

Low-level drivers cannot signal back pressure to upper parts of the stack. Low-level drivers can only cope with the I/O requests they are given but cannot give hints to the upper layers to help with request pacing. It would be useful if back pressure could be signaled to the upper layers of the system so they could limit their generation of I/O requests or make other choices to assist in whatever pacing the lower layer drivers can do.

This I/O scheduler was implemented as a library of routines that the ATA and SCSI periph drivers called to schedule their I/O in a uniform fashion. This suggests that the functionality could more easily be incorporated into CAM's queuing mechanisms.

The Netflix I/O scheduler categorized the normal I/O into a set of new classes. These new classes each had their own queue. These queues exist outside of CAM's notion of `devq`, so the periph drivers needed to be modified as described to work around this mismatch. Future work would include generalizing `devq` in CAM to allow multiple queues.

CAM's allocation of CCBs is needlessly complex for modern machines. It was optimized for machines of the 1990s with 8–32MB of DRAM where the complexity was critical for good performance. Machines with so little memory are now quite rare, even among embedded systems running FreeBSD. It would make sense to modify this code to trade some memory use for code simplicity.

This I/O scheduler helped Netflix's read latency problems. However, it does not match well the classic definition of an I/O scheduler. It provides no way to allocate resources between users or processes, and it provides no way to prioritize I/O for certain users or applications over other I/O to give, for example, a more pleasant interactive response to users. Quotas to limit IOPS or bandwidth are unimplemented.

The Netflix I/O scheduler provided only static configuration of its limiters, perhaps changed by user land monitoring tools. Dynamic steering of rate limiters would provide fast reaction to the changing performance characteristics of the drive. Since write amplification from garbage collection varies

over time, as shown in [24] and [25], dynamic tuning would increase performance. Static tuning cannot take advantage of the dynamically changing bandwidth capabilities the variation in garbage collection implies.

TRIMs currently are implemented using nearly identical code between the SCSI direct access driver (da) and the SATA/ATA direct access driver (ada). Given the differing queue depths, variation in the quality of implementation for TRIMs, and differences between the semantics of an SATA TRIM and a SCSI UNMAP, it is unclear if it would be beneficial to keep these paths separate or if it would make sense to enshrine the code into a common library. Further research in this area is needed.

To date, testing has been limited to one model of SSD from one vendor. Replicating the results across a number of different vendors and models would help validate the range of applicability of these techniques.

Finally, more types of I/O limiting are desirable, such as rate limiting IOPS or bandwidth. In addition, limiting based not on I/O type — but on the class a user or process belongs to — can be beneficial for some applications. None of these traditional scheduling features were provided. It is unclear if the low level is even the right place for them, as they can be more efficiently implemented at the GEOM layer (though actually using them is quite cumbersome due to geom_sched being implemented as a node rather than a policy engine to GEOM). More research into these areas is advised.

X. Conclusion

Placing I/O scheduling at the lowest level has advantages and disadvantages. At the low level, I/O requests can be reordered and paced to have minimal disruption to the system and existing code. The outliers in read latency can be substantially reduced, giving smoother system performance and a better video streaming experience for the customer. However, the complexity of the code and the difficulty of implementing it are a cause for concern. The improved read latency performance came at a rather substantial reduction to the write performance. The benefit is small in comparison to the effort. The work has been worth the effort to Netflix. We've been able to cut the read latency substantially without unduly affecting write throughput. While this specific profile change may be of limited interest, the dramatic results from relatively simple measures strongly suggests further investigation. Reducing the harmful effects of write amplification may be of interest to a wide range of applications. Learning to ameliorate the performance degradation that happens in SSDs and other NAND flash devices could prove quite beneficial.

XI. Updates

An revised version of this paper may be available at http://people.freebsd.org/~imp/bsdcan2015/iosched-v3.pdf. Slides from BSDCan 2015 may be available at http://people.freebsd.org/~imp/bsdcan2015/iosched-slides.pdf. The code for this work can be found at https://svn.freebsd.org/base/projects/iosched. The code has not been updated yet for the changes described in the Recent Changes section below.

XII. Recent Changes

Here's a running log of recent changes to the I/O scheduler that haven't been integrated into the paper yet.

- Added limiters for bandwidth and IOPs. Users can now specify, on a per–disk basis, limits on queue-depth, bandwidth or IOPS. You cannot limit on all these, just one.
- Added timeout call to update quota for each quanta. The bandwidth and IOPS limits are implemented by giving the disk a certain amount of resource to use for each quanta, and then checking to see if there's quota left before doing each I/O. When the timer fires, the scheduler gives each disks its new quota for the time–slice and then sees if there's any pending I/O that can now be scheduled. At the present time, the time slice simply divides the total amount of I/O in a second by the time slice, so if you have very low rates, integer quantization may result in either a lot more or a lot less than desired due to Nyquist sampling limits. The figures in this paper have not been updated to reflect this timeout yet, but it would just be another box that updates things that the can do I/O box would reference to see if I/O is permitted.
- Added the notion of a feedback loop. You can now steer, for example, the write bandwidth limit based on the read load. A control loop to steer the write rate based on the current read latency has been written. Turns out that tuning the control loop has exposed a number of issues.
  - Large quanta cause I/Os to be delayed a long time. In addition `devstat` returns artificially inflated percent busy statistics.
  - Integer quantization effect is much larger.
  - Overhead from timeouts is tiny. Large quanta save little to nothing over small quanta.
  - Drive performance varies second by second, so steering loop needs to steer this quickly. The current EMA stats (exponential moving average) average over tens or hundreds of seconds. This blunts the signals when the performance changes, which retards the reaction of the steering loop.
  - When limiting write bandwidth, write latencies can become quite large.

The addition of steering and dynamic performance tuning has suggested a number of things to explore in the future:

- Feedback to the upper layers would help a lot. When we suddenly have to choke down the write rates, the latency of the writes spikes, and stays spiked because the upper layers keep feeding write requests. Upper layers can make better choices if they know they are dealing with a constrained write (or even read) environment.
- Long latencies suggest a deadline scheduler option would be useful. This is true to solve the problem of read starvation of requests (to help bound latency: the default scheduler has only a very coursed–grained hammer to do

this). However, this would introduce chaining of rules: limit write speed based so it doesn't affect reads, unless the latency is too large, in which case relax the limits in some way. The best way or new limit (if any) when writes are pushed out due to expired timeout for maximum queue latency timed is a poorly understood problem.

- Investigation of making TRIMs optional or advisory so they can be deleted when large numbers are causing delays in processing of other requests. This is an interesting idea, but they can help performance overall (by freeing the blocks in the drive well in advance of writing them again). These effects are hard to measure, so it is hard to study the problem in all but the most carefully controlled environments, let alone automate the decisions based on measured effects.

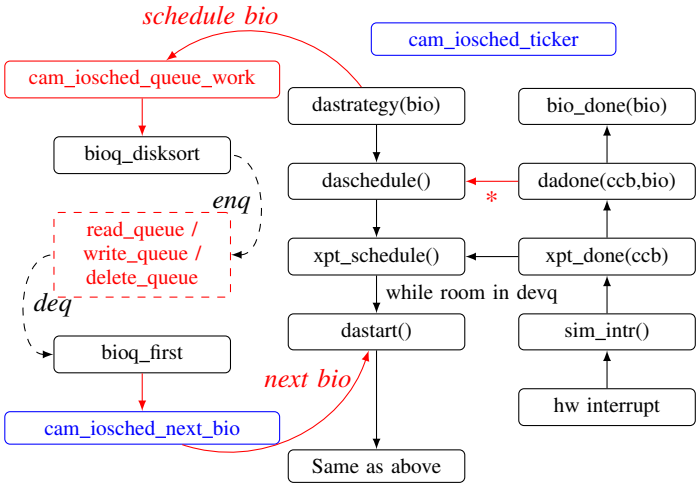

Fig. 8. Recent modifications to disk I/O data path through CAM [23].

References

[1] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD Operating System*. Adison–Wesley, 2015.

[2] P.-H. Kamp, "Disk handling in freebsd 5.x," 2004. [Online]. Available: http://phk.freebsd.dk/pubs/bsdcan-04.slides.geom.pdf

[3] ——, "Geom tutorial," 2004. [Online]. Available: http://phk.freebsd.dk/pubs/bsdcan-04.slides.geomtut.pdf

[4] S. Babkin, *FreeBSD Architecture Handbook*, ch. 12. [Online]. Available: https://www.freebsd.org/doc/en_US.ISO8859-1/books/arch-handbook/scsi.html

[5] J. Gibbs, "Scsi-2 common access method transport and scsi interface module," FreeBSD, 2012. [Online]. Available: http://people.freebsd.org/~gibbs/cam.html

[6] J. Kong, *FreeBSD Device Drivers: A Guide for the Intrepid*. No Starch Press, 2013.

[7] *SCSI-2 Common Access Method Transport and SCSI Interface Module*, ANSI Std. X3.232-1996, 1996.

[8] P. J. Dawidek, *FreeBSD online Manual*, g_bio(9), November 2006. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=g_bio&sektion=9

[9] L. Rizzo and F. Checconi, "Geom_sched: A framework for scheduling within geom," 2009. [Online]. Available: http://www.bsdcan.org/2009/schedule/attachments/100_gsched.pdf

[10] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance," 2008.

[11] D. Dumitru, "Understanding flash ssd performance," 2007. [Online]. Available: http://managedflash.com/news/papers/easyco-flashperformance-art.pdf

[12] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, pp. 1–15, 1992.

[13] X. yu Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," 2009.

[14] *Information technology - ATA/ATAPI Command Set - 3 (ACS-3)*, ANSI X3T13 Working Group Working Draft T13/2161-D, Rev. 5, October 2013. [Online]. Available: http://www.t13.org/Documents/UploadedDocuments/docs2013/d2161r5-ATAATAPI_Command_Set_-_3.pdf

[15] *Information technology - SCSI Block Commands - 3 (SBC-3)*, ANSI X3T10 Working Group Working Draft T10/1799-D, Rev. 25, October 2010.

[16] "Samsung 850 pro ssd review: 3d vertical nand hits desktop storage," 2014. [Online]. Available: http://www.tomshardware.com/reviews/samsung-850-pro-ssd-performance,3861.html

[17] "Netflix accounts for 1/3 of nightly home internet traffic, apple's itunes takes 2 percent," November 2013. [Online]. Available: http://appleinsider.com/articles/13/05/14/netflix-accounts-for-13-of-nightly-home-internet-traffic-apples-itunes-takes-2

[18] "Netflix, youtube gobble up half of internet traffic," November 2013. [Online]. Available: http://www.cnet.com/news/netflix-youtube-gobble-up-half-of-internet-traffic/

[19] "Global internet phenomena report," November 2014. [Online]. Available: https://www.sandvine.com/trends/global-internet-phenomena/

[20] "Content delivery summit," 2013. [Online]. Available: http://blog.streamingmedia.com/wp-content/uploads/2014/02/2013CDNSummit-Keynote-Netflix.pdf

[21] A. Riska and E. Riedel, "Disk drive level workload characterization," Usenix Association, pp. 97–102, 2006. [Online]. Available: http://www.cs.ucsb.edu/~chong/290N-W10/riska.pdf

[22] "Bill swerski's super fans: Da bulls," May 1991. [Online]. Available: https://screen.yahoo.com/bill-swerskis-super-fans-da-000000191.html

[23] "Freebsd source code 10.1 release," FreeBSD Project. [Online]. Available: https://svnweb.freebsd.org/base/release/10.1.0/sys/

[24] "Why solid-state drives slow down as you fill them up." [Online]. Available: http://www.howtogeek.com/165542/why-solid-state-drives-slow-down-as-you-fill-them-up/

[25] "Understanding ssd performance," January 2012. [Online]. Available: http://www.snia.org/sites/default/files/UnderstandingSSDPerformance.Jan12.web_.pdf