Falcon: Scaling IO Performance in Multi-SSD Volumes

Pradeep Kumar and H. Howie Huang, *The George Washington University*

https://www.usenix.org/conference/atc17/technical-sessions/presentation/kumar

This paper is included in the Proceedings of the
2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Falcon: Scaling IO Performance in Multi-SSD Volumes

Pradeep Kumar   H. Howie Huang
The George Washington University

Abstract

With the high throughput offered by solid-state drives (SSDs), multi-SSD volumes have become an attractive storage solution for big data applications. Unfortunately, the IO stack in current operating systems imposes a number of *volume-level* limitations, such as per-volume based IO processing in the block layer, single flush thread per volume for buffer cache management, locks for parallel IOs on a file, all of which lower the performance that could otherwise be achieved on multi-SSD volumes. To address this problem, we propose a new design of *per-drive IO processing* that separates two key functionalities of IO batching and IO serving in the IO stack. Specifically, we design and develop *Falcon*[1] that consists of two major components: *Falcon IO Management Layer* that batches the incoming IOs at the volume level, and *Falcon Block Layer* that parallelizes IO serving on the SSD level in a new block layer. Compared to the current practice, Falcon significantly speeds up direct random file read and write on an 8-SSD volume by 1.77× and 1.59× respectively, and also shows strong scalability across different numbers of drives and various storage controllers. In addition, Falcon improves the performance of a variety of applications by 1.69×.

1 Introduction

The demand of high-performance storage systems is propelled by big data applications that need high IO throughput for processing massive data volumes. Flash-based solid-state drives (SSDs) provide an attractive option compared to hard disk drives for such applications, due to their high random and sequential performance. As a common practice, multiple SSDs are increasingly deployed to support a wide variety of applications such as graph analytics [23, 50, 20, 51, 31, 26], machine-learning [21, 30], and key-value stores [11, 25]. In this work, we especially use a number of graph analytics systems as motivating examples to illustrate the drawbacks of existing approaches.

[1]This system is named after the Millennium Falcon in the Star Wars, “the fastest ship in the galaxy”.

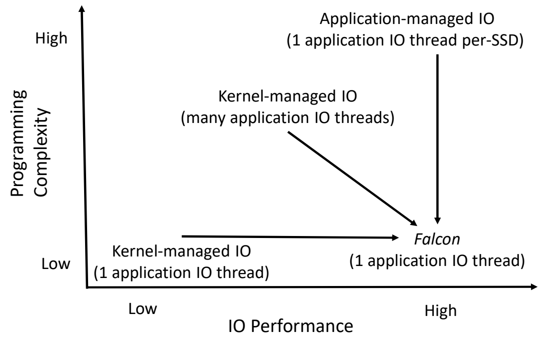
High
Low
Programming Complexity
Application-managed IO
(1 application IO thread per-SSD)
Kernel-managed IO
(many application IO threads)
Kernel-managed IO
(1 application IO thread)
Falcon
(1 application IO thread)
Low
High
IO Performance

Figure 1: Falcon aims to achieve both ease of programing and high IO performance.

To take advantage of high performance of SSDs, throughput-sensitive applications either utilize an application-managed or kernel-managed IO approach as illustrated in Figure 1. In the first case of application-managed IO, prior projects such as SAFS [49], FlashGraph [50], and Graphene [23] require the application developer to explicitly distribute the data among multiple files, each hosted in independent SSDs. In this case, there is no abstraction of a volume, and one application IO thread is dedicated to each SSD. Clearly, such a framework is very complex as applications need to be aware of data partitioning, and determine which application IO thread should perform the IO at any particular instance.

On the other hand, for kernel-managed IO, applications can enjoy the benefits of both volumes and batched IO interfaces provided by the operating system, e.g., Linux AIO (asynchronous IO), Solaris KAIO and Windows Overlapped IO. Such interface allows the applications to submit multiple IOs within a single system call, which provides a clear advantage of ease of programming. However, because IO functionality is limited to just one application IO thread, the combination of kernel-managed IO and a volume, be it created by Linux (e.g., *md*, *lvm*), FreeBSD (e.g., *geom*) or hardware RAID, would fail to saturate the aggregate bandwidth of multiple SSDs.

To mitigate this problem, the applications can spawn a number of dedicated application IO threads to serve the requests in parallel. Several existing projects adapt this

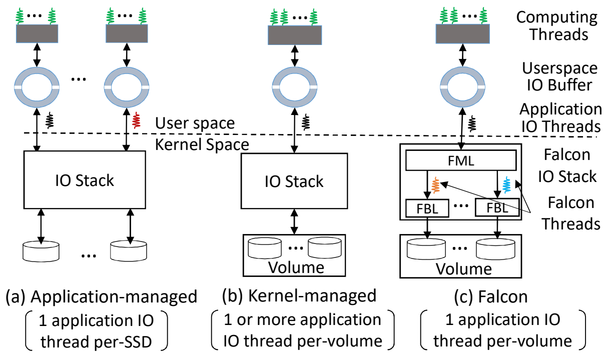

Figure 2: (a) Applications explicitly manage data and IOs on each SSD. There is no volume; (b) Applications relies on the kernel to manage the data on the volume; (c) Falcon maintains the abstraction of volumes and utilizes specialized kernel threads, called Falcon threads, to parallelize per-drive processing for high throughput.

approach, including GridGraph [51] and G-Store [20]. Unfortunately, managing multiple application IO threads using a thread pool is again complicated. And in many cases, this approach does not achieve the expected goal due to the limitations in IO subsystems [28]. For example, many file systems (e.g., ext4 [10]) apply a per-file inode lock, which prevents scalable random read or write from a single file, irrespective of how many application IO threads are employed. So is the case for buffered write where a single kernel thread per volume is responsible for flushing the dirty buffer cache to the volume, limiting the write throughput that could potentially be achieved.

In this work, we strive to achieve the combined benefits of both approaches, that is, delivering high performance IO on a multi-SSD volume while providing ease of programming to the application developers. To this end, we design and develop *Falcon* whose workflow, as shown in Figure 2, presents a new design of *per-drive IO processing* on multi-SSD volumes. The key insight is the separation of the two functionalities of *IO batching* and *IO serving* in the IO stack. The former batches and classifies the incoming IOs at the volume level, and is performed in Falcon IO Management Layer (FML). Meanwhile, the latter serves the IOs in parallel to the SSDs, and is performed in Falcon Block Layer (FBL) using a specialized kernel thread, called Falcon thread.

In particular, FBL provides two new techniques: (1) *per-drive IO sort and neighbor merge*, which limits the scope of sort operations to each SSD and merge to neighboring requests. In contrast, the Linux IO merge algorithm unnecessarily traverses every IO request for all the member SSDs. And (2) *dynamic tag allocation*, which assigns request tags, a limited hardware resource, at runtime. This helps to reduce the unpredictable blocking in the IO stack, and provide a better mechanism to control the number of active IOs in the pipeline, which is applicable across different storage technologies and vendors.

As a result, Falcon allows a dedicated application IO thread to saturate the multi-SSD volume. Thus developers can concentrate more on algorithmic optimizations, without worrying about the complexity of managing multiple application IO threads and SSDs. In contrast, Linux follows *per-volume* approach of mixing IO batching and IO serving tasks in the block layer, where the sequential IO processing and round-robin dispatch lead to many inefficiencies on multi-SSD volumes, and limit the parallelism that could otherwise be achieved.

We have evaluated Falcon with a number of microbenchmarks, real applications, and server traces. Falcon shows strong scalability across different numbers of SSDs, and several different storage controllers. On an 8-SSD volume, Falcon significantly speeds up direct random read and write throughput on an ext4 file by 1.77× and 1.59× respectively, buffered random write by 1.59×, and shows consistent performance for various stripe size configurations. In addition, Falcon speeds up graph processing, utility applications, filebench and trace replay by 1.69×. Lastly, it is important to note that with the new block layer, Falcon is able to saturate a non-volatile-memory-express (NVMe) SSD, delivering 1.13× speedup over the native Linux.

The remainder of the paper is organized as follows. Section 2 presents background on volume management and its interaction with the block layer, as well as how an IO request traverses through various layers. Section 3 quantifies the challenges arising due to per-volume philosophy of Linux IO stack, and presents an overview of Falcon architecture. Section 4 and 5 present the design and implementation of Falcon components. We evaluate the performance of our techniques in Section 6, discuss related works in Section 7, and conclude in Section 8.

2 Background

In this work, while we mostly use Linux to describe the background on the volume management and the block layer; it is worth noting that this IO workflow is generic in nature and many operating systems implement a similar mechanism. Nevertheless, our design and implementation have been influenced by Linux-specific techniques.

In particular, we compare to the Blk-mq [1] block layer which has shown better scaling than the single-queue block layer. Also, most of our discussions pertain to single application IO thread using batched IO interfaces such as Linux AIO. Many kernel daemons such as *pdflush* and *kjournald* submit IO internally in a way similar to batched IO interface. Specifically, pdflush daemon manages the page cache, and has only one dedicated kernel thread per volume to write the dirty pages to storage. There is no pdflush thread to manage the read, and it hap-

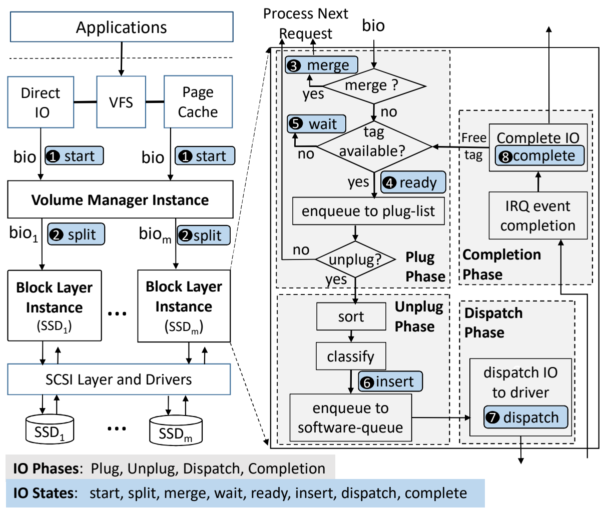

Figure 3: Left: Linux IO stack and the interaction between the volume manager and the block layer. Right: the block layer instance and the detailed IO flow. IO processing happens sequentially, while dispatch happens in a round-robin fashion.

pens directly in the context of the IO issuing thread. Figure 3 shows the inner working of a multi-SSD volume.

Volume Management and Block Layer. An instance of the block layer is associated to a block device, which is associated to a single drive such as an SSD. The volume management layer is used to map several physical block devices into a single virtual block device (e.g., md or lvm). In this layer, IO requests are represented as an object of *block IO* (bio for short) structure. The job of the volume manager instance is to break the incoming bio object into multiple (smaller) bio objects destined for member drives, depending on the IO size and the stripe size of the volume, as discussed next. The original IO is completed only when all the split IOs to different drives are completed.

IO Flow and States. Figure 3 also shows the flow of an IO request from submission to completion within the block layer. For simplicity, we group the process to four phases: *plug*, *unplug*, *dispatch*, and *completion*. IO batching, merge, and tag allocation are performed in the plug phase. IO batching provides an opportunity to merge incoming IOs to take advantage of higher sequential throughput. Also, SSDs provide higher throughput for batched IOs due to parallelism at the hardware level, where more than one IO can be fetched in parallel. Next, sort and classify operations are performed in the unplug phase, IO requests are dispatched to SSDs in the dispatch phase, and IO completion is performed in the the completion phase where various resources are freed.

In each phase, the IO request advances across various states as different tasks are performed on it. As we will show later, one may use the states to track the IO, and find out the time spent by an IO request in different phases for performance profiling.

As soon as an IO request enters the kernel, it is converted to a *struct bio* object and assumes the ❶*start* state. In the case of a multi-SSD volume, the volume manager splits the bio object into multiple smaller objects and moves them to the ❷*split* state. For example, for a multi-SSD volume of 4KB stripes, an incoming IO request of 64KB would be divided into 16 bio objects, each containing 4KB IO destined to a specific SSD. Next, a number of block layer instances (one per SSD) handle the incoming IOs as if it were an IO to this particular SSD. For example, in Figure 3, *bio*1 proceeds to the block layer instance of SSD1, *bio*2 to SSD2, and so forth.

These split bio objects enter their block layer instances in a sequential fashion, and the *plug* phase begins. The operation starts with the bio object being checked against existing IO entries of the per-core *plug-list* for merge candidates. As illustrated in Figure 4(a), the plug-list is a private queue to each IO thread, and is used for batching and merging the incoming IO requests. In other words, the plug-list is shared among multiple block layer instances, and used by all member SSDs of the multi-SSD volume. In this case, a thread does sequential processing of all previously split bio objects, and presents several drawbacks, as we will discuss shortly.

If the bio object is merged, then it moves to the ❸*merge* state, and the processing of the next object starts. Otherwise, a request tag will be requested. If a tag is available then the bio object is put inside a unique *struct request* container indexed by the allocated tag, which in turn is queued to the plug-list. This state is called the ❹*ready* state as IO requests are dispatched in this form to the physical drivers later. However, if a request tag were not available, the IO moves to the ❺*wait* state, and the thread blocks waiting for a free tag.

When the number of IO requests in the plug-list reaches a threshold, an *unplug* event happens, and the *unplug* phase starts. In this phase, all the IOs present in the plug-list of this thread are sorted based on the destination drive and block address information. Next, the sorted IOs move to the per-core, per-drive *software queue* of the member drives, and acquire the ❻*insert* state.

In the *dispatch* phase, the IO requests are dispatched in a round-robin fashion from the software queues to the drives in the same thread context, and moves the IOs to the ❼*dispatch* state. If some IOs can not be dispatched, they will be kept in the per-drive *dispatch-queue* (not shown in the Figure 3) for later processing. Lastly in the *completion* phase, when a drive completes an IO, it raises an IRQ event. The IRQ handler will free the resources and move the IO request to the ❽*complete* state, where any waiting thread is woken up.

Request Tag. The request tag is a limited, vendor and technology specific hardware resource [9]. The available tags are either per storage controller or per-drive.

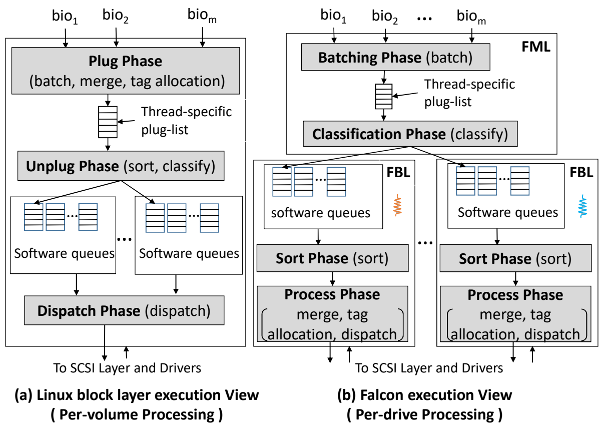
bio1
bio2
biom
Plug Phase (batch, merge, tag allocation)
Thread-specific plug-list
Unplug Phase (sort, classify)
Software queues
Software queues
Dispatch Phase (dispatch)
To SCSI Layer and Drivers
(a) Linux block layer execution View (Per-volume Processing)
FML
Batching Phase (batch)
Classification Phase (classify)
FBL
software queues
Sort Phase (sort)
Process Phase merge, tag allocation, dispatch
(b) Falcon execution View (Per-drive Processing)

Figure 4: (a) IO issuing thread batches the IOs destined to different member drive to same plug-list. The merge, tag allocation, and sort operation being performed in the plug-list is the major cause of inefficiency. (b) Falcon's idea of per-drive philosophy is to postpone the IO serving tasks of the block layer to the drive-specific software-queue. Completion phase is omitted for simplicity.

For example, the Intel SCU technology has 250 available tags [39], but they are shared among four ports of the controller. That is, every connected SSDs will compete for the same tag space. Similarly, the LSI 9300-8i SAS HBA adapter has 10,140 tags, and is shared by all the connected drives. On the other hand, the Intel AHCI SATA controller has only 32 tags per SATA port, which is not shared. In this case, the tag count matches with the drive's internal queue size. For the Samsung 950 pro 512GB NVMe SSD that we use in this work, the tag counts are 1024 per hardware-queue. This specific drive has 8 hardware queues [38], while SATA SSDs have only one hardware queue.

3 Falcon Architecture

In this section, we first describe the insufficiencies of current per-volume processing of Linux IO stack, and present the overall architecture of Falcon.

3.1 Challenges of Per-Volume Processing

Current multi-SSD volumes follow the per-volume processing, that is, IO serving is tied to the plug-list, and is forced to be performed in a sequential manner within a volume. In other words, as shown in Figure 4(a), the plug-list mixes IO requests that actually belong to various member drives within a multi-SSD volume. Moreover, the block layer mingles two unrelated tasks: batching, and merge/tag allocation in its plug phase, and sort and classify in the unplug phase.

To illustrate the problems, we run a revised FIO benchmark [12] on various configurations of multi-SSD volumes. The detailed setup will be presented in Section

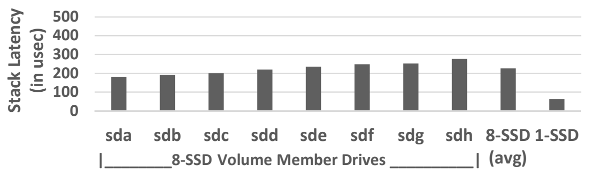

Figure 5: Stack latency of eight SSDs, and average latency

6. In particular, we measure two metrics: the *stack latency* is the time between the start and insert states, while the *device latency* is the time between the dispatch and completion states. We use the former to gauge the software performance, and the latter for device performance.

Insufficiency #1: Lack of Parallelism. As several IO serving tasks are forced to be performed in a single plug-list, the opportunities in parallelizing those tasks are limited. Under the Linux architecture shown in Figure 4(a), merge, tag allocation, and sort tasks lack parallelism, while dispatch happens in a round-robin way.

Figure 5 quantifies this impact on the stack latency of SSDs within a volume. Out of 8 SSDs, the slowest drive (*sdh*) spends at least 55% more time on IO processing (i.e., the stack latency) as compared to the fastest drive (*sda*). Interestingly, the latency increases in the same order of the drives. This is due to the round-robin dispatch where the first drive always gets the highest priority to dispatch followed by the second drive onwards. As a result of this procedure, later drives have to wait even though the requests are ready to be dispatched.

Insufficiency #2: Inefficient Merge and Sort. The current merge algorithm traverses the plug-list of the thread to find the merge candidate for a bio object. It searches all IO requests including those that belong to different drives. But clearly, they should not be considered as candidates at all. Also, in the unplug phase, sorting happens on the same thread-specific plug-list, which again means wasteful processing on irrelevant requests.

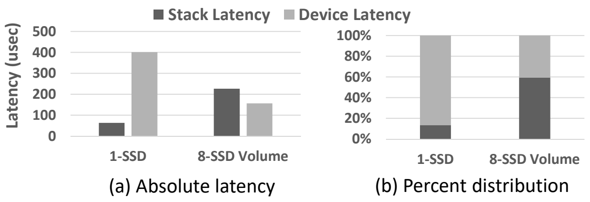

Figure 6: Distribution of the stack latency and device latency, showing absolute and percentage distribution. For 8-SSD volume, stack latency is more than the device latency.

Making matters worse, the IO count in the plug-list is significantly higher for a multi-SSD volume. The plug phase ends only when the merge task finds more than 16 IOs (the per-drive threshold) in the plug-list destined to the same drive. Assuming an uniform distribution of the IOs among all the drives, the total number of the IOs in the plug-list would need to reach 128 for an 8-SSD volume to end the plug phase, as opposed to 16 for 1-

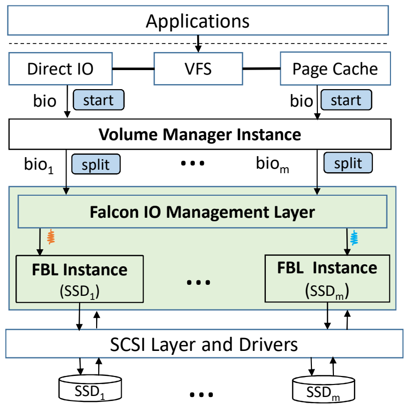

Figure 7: Falcon IO stack highlighting differences with Linux IO stack. A new abstraction in the form of *Falcon IO Management Layer* (FML) is introduced which performs purely IO batching work, while the Falcon Block Layer (FBL) performs IO serving tasks in parallel to FML and to each other.

SSD. As a result, the average processing time spent by an IO thread for the 8-SSD volume is significantly higher, over 3.5× more than 1-SSD (Figure 6(a)), and forces the IO thread to spend 60% time inside the IO stack, pointing to the IO stack as the bottleneck (Figure 6(b)).

Insufficiency #3: Unpredictable Blocking. In between the merge and sort tasks, the tag allocation is forced to be performed in a sequential manner as well. So, when a tag allocation fails for any drive member, the executing thread blocks the whole IO stack waiting for a free tag from that drive. Thus the active IO count present in the Linux IO stack is controlled by the tag count because the blocked IO thread wakes up only when the tag becomes available, i.e. only when an existing IO completes. This blocking is unpredictable, as the tag count varies and can either be storage controller or drive specific.

3.2 Per-Drive Processing in Falcon

Falcon proposes a new approach of the *per-drive philosophy*, which separates the two operations of IO batching and IO serving by regrouping the tasks by their functionalities in new phases, as shown in Figure 4(b). Specifically, only IO batching and classify tasks are performed in the plug-list. And merge, tag allocation, and dispatch tasks move to a new *process phase* and are performed in per-drive software queues, and can easily be parallelized. This reduces the amount of work being done in the plug-list, and hence removes the major bottlenecks.

Figure 7 presents the major components of Falcon. In particular, the new batching and classification phases are performed in the *Falcon IO Management Layer* (FML for short), while the sort phase along with the process and completion phases are performed in the *Falcon Block Layer* (FBL). Moreover, the FML also spawns Falcon threads for parallel IO serving across FBL instances.

Block Layer Features	Linux 1-SSD	Linux Multi-SSD	Falcon Volume
Parallel processing	NA	✗	✓
Per-drive sort	✓	✗	✓
Neighbor merge	✗	✗	✓
Dynamic tag management	✗	✗	✓

Table 1: Falcon's per-drive processing

We summarize the differences between Falcon and Linux IO stack in Table 1. For example, the Linux Blk-mq architecture allows per-drive sort for 1-SSD system, but it fails to provide the same functionality to multi-SSD volumes. In contrast, the separation of functionalities allows the Falcon to keep the per-drive philosophy intact in its FBL block layer, as the IO serving operations are performed in the per-drive software queue. In addition, dynamic tag allocation in FBL removes the tag allocation from the plug phase and moves it to just before dispatch. This provides a more uniform and predictable criterion to control the outstanding IOs in the IO stack pipeline.

4 Falcon IO Management Layer

Falcon IO Management Layer (FML) is the new abstraction between the volume manager and the block layer. It performs IO batching, and creates a Falcon thread for each FBL instance to parallelize per-drive processing. Figure 8 presents the IO flow in the management layer.

4.1 IO Batching

IO batching is performed in two phases: batching and classification. FML starts its batching phase as soon as a new (split) bio arrives, and pushes the object into the plug-list of the thread. Next, the volume manager sends the next bio object of the original IO request. If there are no more objects, the volume manager processes the next request from the batched IO interface. This bio object is again enqueued to the same plug-list by the FML layer.

As this process progresses, an unplug event will occur. At this point, the current batching phase stops, and the classification phase begins, thereby, all the bio entries in the plug-list are classified based on destination drives.

Batching and classification tasks are performed using bio objects in Falcon, as opposed to request containers in Linux, hence we add *prev* and *next* pointers to the bio structure, so that the bio objects can be chained in the doubly linked-list plug-list.

Additionally, at the end of the classification phase, all the IOs have to be enqueued to the per-core per-drive software queue. However, software queues are protected by spin locks, and acquiring them becomes mandatory each time an IO need to be enqueued. To this end, we collect the requests in temporary per-drive queues during the classify operation. At the end, all bio objects from the

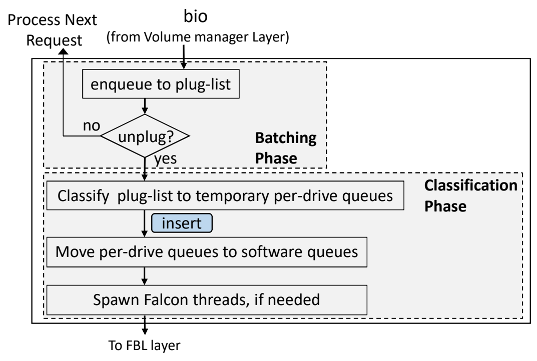
Process Next Request
bio
(from Volume manager Layer)
enqueue to plug-list
no
unplug?
yes
Batching Phase
Classify plug-list to temporary per-drive queues
insert
Classification Phase
Move per-drive queues to software queues
Spawn Falcon threads, if needed
To FBL layer

Figure 8: IO flow in Falcon IO Management Layer. The plug-list is used only for batching operations. Merge, sort, and tag operations are no longer performed in here.

temporary queues move to respective software queues, thus acquiring the spin lock only once.

4.2 Enabling Parallel Processing

At the end of the classification phase, all the IOs reside in the *per-drive* software queue, and hence the FBL instances can perform IO serving tasks in parallel, which is not possible in Linux. To this end, FML spawns one kernel thread per participating FBL instance for this group of batched IOs. We call each thread a *Falcon thread*, and is responsible for IO serving tasks.

In our implementation, the Falcon threads are spawned using Linux's kblockd workqueue object. However, the CPU affinity of a kworker is decided by the thread that requests a kworker. Should all the Falcon threads share the same core as the IO issuing thread, it would defeat the purpose of parallel IO serving. To address this problem, we modify the workqueue API invocations so that we can use the CPU affinity of each Falcon thread to pin them to different cores. Ideally it should be NUMA-aware, that is, on a core of the socket that hosts the corresponding storage controller adapter. We manage this information inside *hardware context* object of each drive.

The completion phase is also executed in the same core to which the Falcon thread is pinned. As the CPU core information is now available, the IRQ handler can send an inter-processor interrupt (IPI) to this core to perform the completion phase, or execute directly if the IRQ is received on the same core as that of the Falcon thread.

The job of completion phase is three-fold: freeing up bio objects, request tag and other resources; waking up any thread that is waiting for IO completion; and requesting a Falcon thread to resume processing if the internal queue of the drive becomes full at the last dispatch. In performing those tasks, a completion thread accesses those data structures that are set by the Falcon thread. Hence by running the completion on the same core, Falcon avoids the cache migration of those objects, making completion a cache friendly phase.

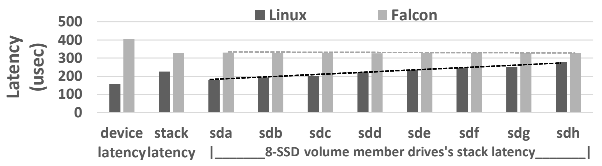

Figure 9: Drive and stack latency comparison of 8-SSD volume. Falcon is able to make the stack latency uniform as well as smaller than the device latency.

4.3 Unplug Criteria

One implication of separating the merge task from the batching task is removal of the unplug criteria from the plug phase. The unplug event is raised when the number of requests evaluated for merging goes beyond a threshold in the plug phase. As we have mentioned earlier, the criteria varies depending on the IO distribution to each drive, and the maximum value of the threshold is 128 for 8-SSD volume in a uniform distribution. To achieve more predictable unplug events, Falcon utilizes two new thresholds (*low watermark* and *high watermark*). Simply put, we maintain the IO requests in the plug-list and finish the batching phase when the count reaches beyond a threshold.

Increasing the watermark by too much would increase the stack latency, and as a result the device would remain idle because more IO requests are still being processed. On the other hand, lowering the watermark may potentially reduce the benefit of batching. An equilibrium is desired so that the drives are kept busy as long as there are sufficient number of IO requests. Given a typical internal queue size of 32 for an SSD, we choose the product of device count and this queue size as the high watermark value, e.g., 256 for an 8-SSD volume.

Figure 9 plots the stack latency for Falcon using the high watermark. Since the ordering of IO state has changed, here we measure the new stack latency as the time between the start to ready phase. One can see that Falcon is able to achieve similar stack latency for different drives, compared to a large variance in Linux. Specifically, Falcon achieves around 320 microseconds, smaller than 404.7 microseconds device latency from the SSDs. As such, the stack and drive latency are nicely balanced.

It should be noted that the device latency is a function of the queue depth, i.e. a busy SSD will have higher device latency than one that is lightly loaded. In Linux's case, the device is operating at lower queue depth, as the application IO thread is not able to dispatch enough requests. In contrast, thanks to parallelism, Falcon threads in Falcon are able to dispatch more IOs to SSDs, and keep them busy all the time. As a result, the stack latency is smaller than the device latency. So, even with higher stack latency than Linux, Falcon is able to get higher throughput as we will show in Section 6.

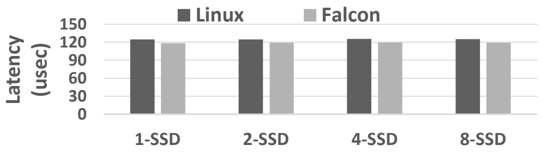

Figure 10: IO latency study of various multi-SSD volume

Latency. Falcon uses the low watermark to facilitate latency-sensitive applications, where only fewer IOs are submitted. The idea is to avoid an extra context switch when the IO demand is low. Here we take current value of 16 requests per drive as the basis, and set the low watermark as the product of this value and the drive count, e.g., 128 IOs for 8-SSD volume. It should be noted that if fewer IOs are submitted in a batched IO interface or just one IO using POSIX IO interface, the batching phase does not wait for more incoming IOs, and an unplug event occurs at the end of the submission. For 1-SSD system, Falcon always performs synchronous IO serving. Note that it is also possible to let the users choose both high and low watermarks depending on their need.

For such applications, IO serving will happen in the context of the IO issuing thread as the plug-list would not cross the low watermark. Figure 10 shows that Falcon improves the IO latency (from the application perspective) by nominal 3% for various multi-SSD volumes (RAID0, 4KB stripe size) when just one IO of size 4KB (POSIX IO) is active in the whole IO pipeline.

5 Falcon Block Layer

Falcon Block Layer (FBL) is the new block layer that performs the IO serving tasks. FBL instances receive unsorted bio objects in their per-core, per-drive software queues. Compared to the existing approach where most of the operations happen in the per-thread plug-list, our approach enables per-drive processing, which can be divided into three phases (sort, process, and completion) as shown in Figure 11.

5.1 Per-Drive Sort and Neighbor Merge

Mechanism. The software queue is a per-core queue, so a single drive has many associated software queues, one for each CPU core. Hence the sort phase first aggregates the bio objects from all of the software queues of the drive in a private queue (a doubly linked-list), and then performs sorting on it. This results in all neighboring IOs being adjacent to each other in the private queue, thus only a neighbor merge is required in the process phase, which happens as follows.

A Falcon thread removes the first bio object from the private queue, allocates a tag, and puts it inside a request container object indexed by the tag. Then, the merge task checks the next bio entry in the private queue to see if it

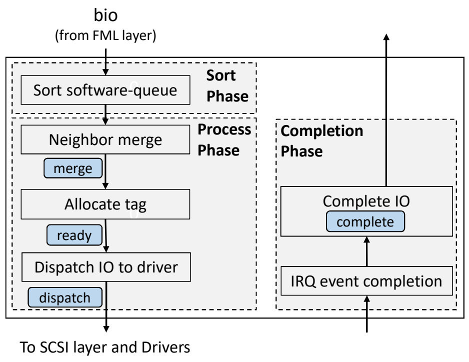

Figure 11: Falcon Block Layer IO flow, and states

can merge with the current container. If it succeeds, the next entry in the queue will be tried for merge. Otherwise, the container object will be dispatched. The process goes on till either all entries are dispatched or the internal queue of the drive becomes full.

The internal queue of an SSD may become full due to its limited size. In this case, all the requests need to be preserved to be dispatched later. The last request container is kept in the *dispatch-queue*. We introduce a new per-drive queue, called *bio-queue*. Its role is similar to dispatch-queue, but keeps the remaining bio entries of the private queue. Later, when triggered, IOs are first dispatched from the dispatch-queue followed by the bio-queue. The separation of IOs in different queues are required as the IOs are in different states. The order maintains the prior behavior of request dispatch.

The sort task requires multiple pass over IOs in the private queue which collects bio objects from software queues. It is possible that the sort task might dominate the overall processing in the IO stack. We leave the investigation of a new data-structure for queues as future work.

Advantages. The new merge technique is very simple and presents several benefits. First, sorting runs efficiently with less CPU usage due to smaller per-drive sort space. Second, the merge algorithm needs to evaluate its neighbor requests only as they are already sorted, which reduces the CPU utilization further. Third, since merge happens on the private queue containing IOs from software queues of the different cores, one can automatically achieve merging across multiple IO issuing threads.

It is worth noting that efficient sort and neighbor merge are generic improvements to Linux IO stack. For example, single application IO thread is not able to saturate an NVMe SSD (Samsung 950 pro 512GB NVMe) in Linux [18]. However, as shown in Figure 12, Falcon can saturate it (1375 MB/s) for random read workload using FIO benchmark. In this case, Falcon does synchronous IO serving by default.

Linux Blk-mq layer treats NVMe SSDs differently from SATA SSDs. Only two incoming IOs are consid-

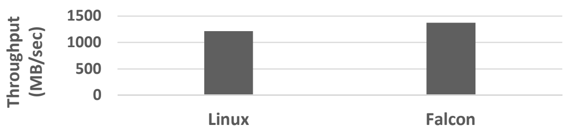

Figure 12: Impact on NVMe SSD for random read

ered for merging for NVMe SSDs, and if that fails, the older IO is dispatched and the most recent is kept in the plug-list. In contrast, Falcon does not differentiate between NVMe and SATA SSDs. In this case, per-drive sort and neighbor merge makes the batching efficient, without any sacrifice on latency.

5.2 Tag Management

Problems. Controlling the active IO count in the IO pipeline based on a vendor and technology specific tag count often leads to unpredictable results for random IOs in multi-SSD volumes. Random IOs are inherently skewed towards some drives within any small time duration. This leads to unfairness in the tag allocation for member drives, resulting in compromised performance for Intel C602 AHCI SATA III connected volume as shown in Figure 13(a).

The reason is that after allocating 32 tags for a SATA SSD, the IO thread would block for an additional tag for the SSD, even if other SSDs might have available tags. In a skewed IO distribution case, 2-SSD SATA volume can only maintain less than 40 active IOs in the IO pipeline against the available tag of 64 as shown in Figure 13(b), resulting in the throughput drop. When there are sufficient number of tags such as LSI HBA which has over 10,000 tags, the volume does scale on multiple SSDs on both Linux and Falcon.

Dynamic Tag Allocation. To provide a predictable behavior, a uniform count of active IOs must be maintained in the IO pipeline, regardless of the storage technology or vendor. Therefore, Falcon performs the tag allocation *dynamically*, i.e. only if a dispatch is required in the process phase, as shown in Figure 11. The main benefit is improved queue utilization because more IOs are allowed to reside in the IO pipeline without acquiring a tag. This offsets the skewness of random IO distribution.

Figure 13(a) compares the throughput scaling of Linux and Falcon for a 2-SSD volume connected using Intel C602 AHCI SATA III. The throughput improvement is due to improved tag utilization of both the member drives as shown in Figure 13(b). The drop in tag usage between 5–19 seconds is due to highly skewed workload distribution (random read in FIO benchmark), where only one drive's internal queue is fully utilized. However, Falcon can still get close to 2× IO performance improvement over 1-SSD volume. The technique results in 52% and 23% improvement in random read and write respectively, saturating the volume completely.

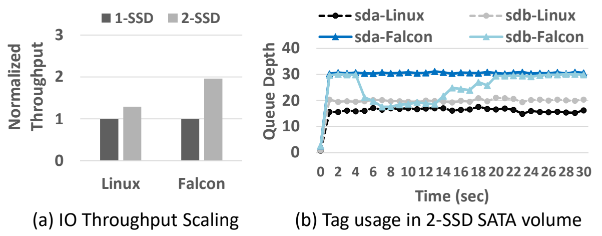

Figure 13: Impact of dynamic tag allocation on 2-SSD SATA volume connected using the Intel C602 controller

Back Pressure. Tag allocation serves as a back pressure point in the Linux IO stack. That is, if the number of in-flight IOs were to increase beyond the available tags, the IO issuing thread would go to sleep and stop submitting new IOs. Moving the tag allocation from the plug phase also removes the back pressure point in Falcon, and hence the IO issuing thread could potentially keep submitting as many requests as it can, and consume a lot of system resources such as memory.

To address this problem, Falcon proposes a per-drive limit. When the number of IOs increases beyond a high-pressure point, the thread stops IO submission in FML and sleeps. The thread will become active again when the number of requests drops below a low-pressure point in the whole IO stack, thus controlling the in-flight IO count in the whole IO pipeline. The number of requests in the bio-queue is used to determine the pressure point. For a multi-drive volume, the pressure point is equal to the product of per-drive pressure point and drive count.

The pressure point is a different threshold than the watermark. The former is about when to block IO processing thread, while the latter is used to decide when to do synchronous or parallel dispatch.

6 Experiments

The machine used for the experiments has dual-socket of Intel Xeon CPU E5-2620 2GHz with six cores each, thus total 24 threads due to hyper-threading, and 32GB DRAM. We use eight Samsung EVO 850 500GB SSDs connected using LSI SAS9300-8i HBA which supports SATA III SSDs. The system also has a Samsung 950 pro 512GB NVMe SSD with PCI 3.0 interface, two Intel AHCI SATA III ports, four SATA II ports, and four SCU ports which supports SATA II SSDs. We use four Intel 520 120GB SSDs for testing SCU ports.

We run the tests on Linux kernel version 4.4.0 with the Blk-mq block layer [1] (which performs better than the single-queue block layer). The blk-mq architecture has been completely integrated with SCSI layer and other drivers (called scsi-mq) in this kernel. Currently, blk-mq does not have any configurable IO scheduling policy.

We have implemented a prototype of Falcon in about 600 lines of C code with the aforementioned Linux kernel. We use the *md* software as the volume manager with default stripe size of 4KB in a RAID-0 configuration. By default we use raw volumes, and also evaluate ext4 and XFS file system in a number of cases.

6.1 Microbenchmarks

We use a modified FIO in these tests. FIO [12] provides a number of IO engines such as AIO and synchronous POSIX IO and outputs a number of parameters including throughput, IOPS, and latency. However, FIO spends a lot of time in userspace (more than 35%), thus can not submit IOs as fast as a single application thread can otherwise. To address this problem, we modify FIO to instead simply replay the traces as fast as possible.

Ext4 File Throughput. The per-inode lock on ext4 File System does not allow Linux to saturate the 8-SSD volume even using multiple application IO threads. However, Falcon can saturate the volume using just one application IO thread, as shown in Figure 14. The improvement is due to parallelism at block layer tasks (sort, merge, tag allocation and dispatch). Overall, Falcon achieves 1.77× and 1.59× random read and write throughput compared to Linux on an ext4 file in 8-SSD volume.

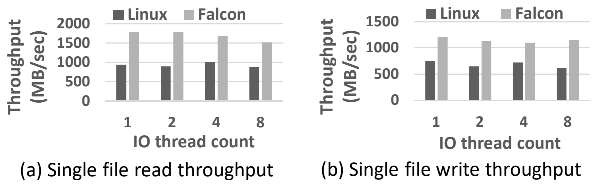

Figure 14: Random IO on an Ext4 file in 8-SSD volume

Buffered Write Throughput. Figure 15 shows improvement in buffered write throughput when 8 application IO threads are doing random write on multi-SSD volumes. Again, Linux is not able to achieve beyond 800 MB/s throughput on 8-SSD system because Linux buffer cache management allows only one *pdflush* thread to write the dirty buffer to the volume. In contrast, Falcon achieves 1.38× and 1.59× improvement compared to Linux in raw 4-SSD and 8-SSD volumes.

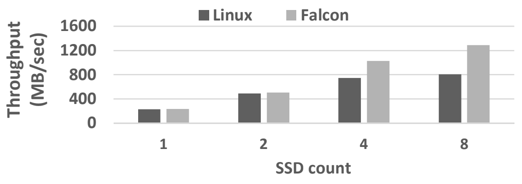

Figure 15: Buffered random write throughput scaling

Varying SSD Count. Figure 16 shows that Falcon delivers performance improvement by 1.92×, 3.65× 6.02× for random read, and 1.86×, 3.34× and 6.29× for random write in 2-SSD, 4-SSD and 8-SSD volumes over one SSD respectively. This clearly indicates that Falcon is scalable when more SSDs are added to the volume.

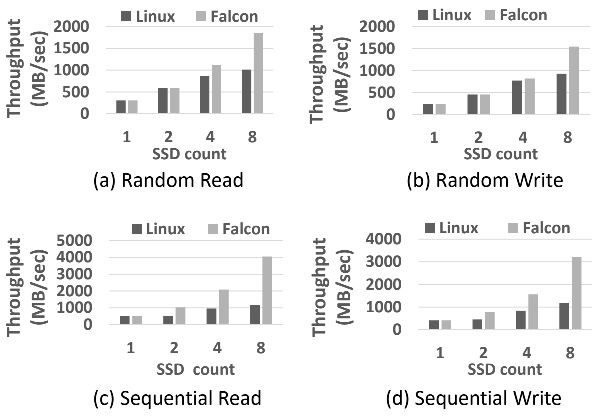

Figure 16: IO Scalability by varying the number of SSDs

For the 8-SSD volume (LSI HBA), Falcon achieves 1.83×, 1.66×, 3.42× and 2.73× speedup for random read, random write, sequential read and sequential write, respectively. When using the SCU controller for 4-SSD volume, Falcon can also achieve 1.25×, 1.08×, 1.59× and 1.85× respectively, again saturating the volume completely.

Varying Stripe Size. Figure 17 shows the random and sequential IO throughput on an 8-SSD volume for a variety of stripe size configurations. Random IO (4KB IO size) is highly susceptible to stripe size configuration as a better IO distribution to all the member drives will maximize the IO, while a skewed distribution would not. Figure 17 shows that 4KB stripe size is the best, while 32KB stripe is the worst for the random IO pattern generated by the FIO. In the best case, Falcon provides 1.83× and 1.66× random read and write throughput respectively.

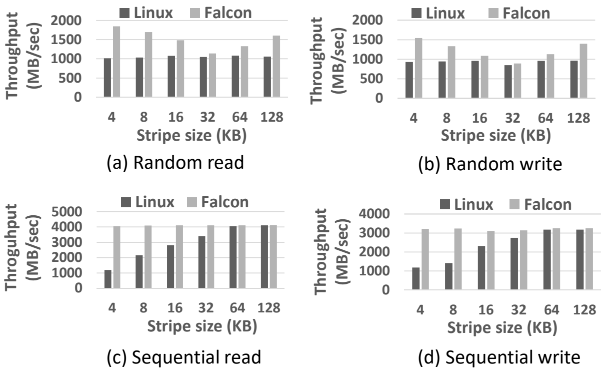

Figure 17: Throughput for various stripe sizes in 8-SSD volume

Sequential IO (64KB IO size) generates uniform IO load to each member drive of the volume. For 4KB stripes, Falcon provides 3.42× and 2.73× sequential read and write throughput respectively compared to Linux. Falcon saturates the 8-SSD volume irrespective of stripe size, while Linux can saturate the volume only when the IO size is smaller than or equal to the stripe size. This is because the Linux block layer is written with the assumption of a single drive. In other words, the

Linux block layer assumes that split has been performed for the IOs larger than 1 MB (maximum IO size in the block layer), and wrongly determines that further merging would no longer be needed on split IOs. In contrast, FBL enables the merge in case of IO split, and enjoys the performance benefit from the full sequential IOs.

Advantage and Scalability. There are two important observations when running Falcon on an 8-SSD volume. First, Falcon saturates the sequential read/write completely (Figure 16). Second, for random read and write tests, Falcon achieves 97.6% and 98.9% throughput of an ideal system, where one application IO thread is dedicated to submit batched IOs to each SSD independently, so that IO skewness is not the concern.

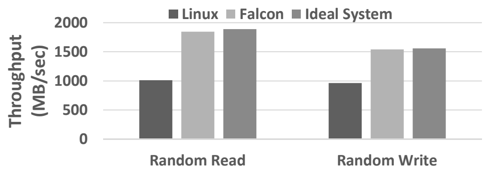

Figure 18: Random read comparsion of Linux and Falcon with an ideal system on an 8-SSD volume.

To understand the maximum random throughput achievable by an application IO thread, we run the same experiment in a null block device. The device is a standard Linux driver without any backup storage, and acknowledges the IOs as soon as received. Falcon can provide up to 3.7GB per second random read, which would be roughly equivalent to the aggregate throughput of 16 SSDs. However, it should be noted that null block device avoids many operations which otherwise were needed for a real volume. We expect that spawning more than one application IO thread will saturate 16 or more SSDs.

6.2 Application Performance

All the tests in this section are performed in XFS File System. For Linux, XFS tends to outperform ext4 for parallel reads as it does not acquire an inode lock. Since Falcon always deploys a single application IO thread, the choice of file system would not matter.

Utility Applications. We choose *copy* and *tar* to show the effectiveness of Falcon. Copy represents parallel data copying between the volume and a 24GB RAM disk. Specifically, *CopyTo* copies the 24GB file from the RAM disk to the volume. *CopyFrom* does the reverse. On the other hand, *Tar* and *Untar* use pbzip2, a parallel implementation of bzip2. As shown in Figure 19(a), Falcon speeds up CopyFrom, CopyTo, Tar and Untar by 1.63×, 2.81×, 1.29× and 1.09× respectively. The benefits are lower for Tar and Untar as they are more CPU intensive.

Filebench. We also run Fileserver (2:1 read/write ratio), Webserver (mostly read with random appends) and Webproxy (read only) personality in Filebench suite of

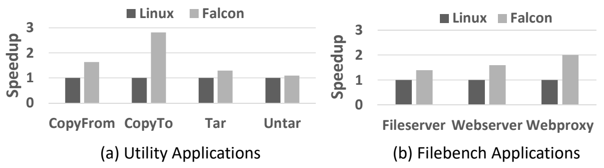

Figure 19: Application performance on 8-SSD volume

benchmarks. Figure 19(b) shows that Falcon performs 1.39×, 1.60× and 2× better than Linux when the benchmarks ran on around 64GB of data.

Graph Processing. We choose G-Store [20], a semi-external graph processing system as a representative use-case for high throughput application to demonstrate the effect of Falcon. In particular, we evaluate four different graph algorithms including breadth-first search (BFS) [7, 14], kCore [33, 29], connected component (CC) [35] and page rank (PR) [2] algorithms. BFS and kCore generate very high random IOs on graph data, while CC and PR require mostly sequential IOs. The experiments are on a undirected kronecker graph of scale 28 and edge factor 16 [14].

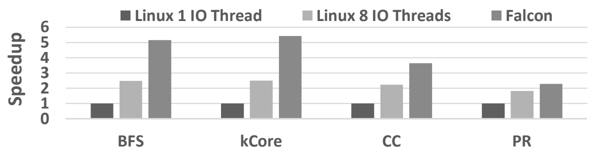

Figure 20: Graph Processing Performance

Figure 20 shows that Falcon significantly speeds up the graph processing by 4.12× and 1.78× compared to using one and eight IO threads in Linux. In particular, Falcon achieves more than 5× speedup for BFS and k-Core compared to using one IO thread in Linux, and 2× improvement over 8 IO threads. In the case of CC and PageRank, Falcon also provides 2 to 3× improvement over using a single IO thread.

6.3 Server IO Traces

We run application traces collected at University of Massachusetts Amherst [41] and Florida International University [19]. Table 2 provides the information on these traces. UM-Financial1 and UM-Financial2 represent the OLTP type applications, while UM-Websearch1 and UM-Websearch2 are websearch traces. On the other hand, FIU-Home, FIU-Mail, FIU-Webuser and FIU-Web-vm represent the traces from home directory, mail, web user, and webmail proxy and online course management system. The traces contain the lower-level IOs, i.e., at the logical block address. Almost all the write operations are caused by *kjournald* or *pdflush* daemons which run in the kernel space. These daemons behave more like a batched Linux AIO interface.

We replay the traces by submitting IOs as fast as the system allows. Figure 21 shows *Falcon* can extract

Trace Name	Read (%)	IO size range	Size (GB)
UM-Financial1	23.16	512B - 16715KB	17.22
UM-Financial2	82.34	512B - 256.5KB	8.44
UM-Websearch1	99.98	512B - 1111KB	15.24
UM-Websearch2	99.98	8KB - 32KB	65.82
FIU-Home	1.00	512B - 512KB	34.58
FIU-Mail	8.58	4KB - 4KB	86.64
FIU-Webuser	10.33	4KB - 128KB	30.94
FIU-Web-vm	21.8	4KB - 4KB	54.52

Table 2: IO traces summary

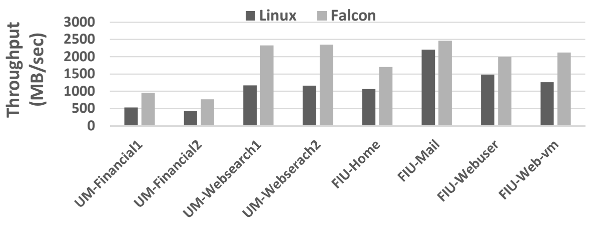

Figure 21: Trace replay throughput on 8-SSD volume

more throughput for all IO traces. Overall Falcon performs 1.67× better over Linux. UM-Financial1, UM-Financial2 has small throughput as they do many 512 byte random IOs. FIU-Home traces are almost write-only and random IO. Others are mix of random and sequential IOs.

7 Related Work

Prior works [4, 5, 46, 37, 43, 45, 48, 17, 36, 47] mostly aim to improve the IO stack for one drive by proposing changes in the IO stack and/or hardware, leaving behind a number of issues pertaining to multi-SSD volumes. Different from these approaches, we focus on multi-SSD volumes and aim to achieve both the IO scalability and the ease of programming.

Application-managed IO approach [49, 50, 23] partitions a file in different SSDs and proposes a userspace abstraction to aggregate the file content. As a result, it introduces a lot of complexity in the application, and lacks support of POSIX file system [49]. Also, there are application level restrictions, such as only integer number of processing cores for each SSD (e.g., one compute thread for each SSD in Graphene [23]).

Various works [40, 27, 22, 34] have identified the importance of IO stack optimization, and have proposed various changes including the block layer to accelerate application performance. A nice description of the time spent on different layers of the IO platform has been analyzed in [13]. Problems with sub-page buffered write is identified [3] and techniques have been proposed to improve the performance. Wang et al. [44] propose fairness and efficiency in tiered storage system. Our work is in-tune with these efforts and especially identifies improvements in the IO stack for multi-SSD volumes.

Linux kernel developers have made many improvement in various locking semantics [15, 16, 24], however, locking in parallel IO from the same file still remains an issue as observed by Min et al. [28] in a many-core system. An enhanced storage layer has been proposed in [8] that exports information to file systems to bridge the information gap. Our system utilizes such information within a new layer (FML) to improve the performance in multi-SSD volumes.

Storage area network (SAN) solutions provide aggregated SSDs as block device services, and scale in terms of multiple clients. However, the local file system at client side will still have all the constraints that we have discussed. Chen et al. [6] have developed new batched IO interface, and integrated it with NFS compound procedure using a userspace NFS client. In such cases, Falcon will become a natural choice of IO stack to provide a better client-side IO stack to take advantage of the faster SAN/NFS storage.

Big data applications such as WiscKey (a key-value store) [25] deploy a complex mechanism of a thread pool to serve IOs in one SSD. To scale to multiple SSDs, they need to fine-tune the number of threads. Falcon will enable such applications to move towards just one thread, and can saturate more SSDs if AIO interfaces are used, thereby providing simplicity and scalability both.

Lastly, many graph applications [21, 32] bypass the random IO problem in multi-SSD volumes by fetching the whole data in each iteration. However, Vora et al. [42] show that the performance of many graph algorithms can be improved by doing selective random IO. We believe that Falcon will become a platform of choice for many IO-intensive applications including graph analytics.

8 Conclusion

In this work, we have identified that the separation of two IO processing tasks, i.e., IO batching and IO serving in the block layer, holds the key to improve the throughput in multi-SSD volumes. To achieve this goal, *Falcon* proposes a new IO stack to enforce per-drive processing that improves the IO stack performance and parallelizes the IO serving tasks. Compared to current practice, Falcon significantly accelerates a variety of applications from utility applications to graph processing, and also shows strong scalability across different numbers of drives, and various storage controllers.

9 Acknowledgments

The authors thank the USENIX ATC'17 reviewers and our shepherd Sorav Bansal for their suggestions. This work was supported in part by National Science Foundation CAREER award 1350766 and grant 1618706.

References

[1] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 22:1–22:10, New York, NY, USA, 2013. ACM.

[2] S. Brin and L. Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *Proceedings of the Seventh International Conference on World Wide Web 7*, Amsterdam, The Netherlands, 1998.

[3] D. Campello, H. Lopez, R. Koller, R. Rangaswami, and L. Useche. Non-blocking Writes to Files. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 151–165, Santa Clara, CA, Feb. 2015. USENIX Association.

[4] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society.

[5] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, New York, NY, USA, 2012. ACM.

[6] M. Chen, D. Hildebrand, H. Nelson, J. Saluja, A. S. H. Subramony, and E. Zadok. vNFS: Maximizing NFS Performance with Compounds and Vectorized I/O. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 301–314, Santa Clara, CA, 2017. USENIX Association.

[7] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems*.

[8] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 177–190, Berkeley, CA, USA, 2002. USENIX Association.

[9] Disk Controller Queue-Depth. `http://www.yellow-bricks.com/2014/04/17/disk-controller-features-and-queue-depth/`, 2016.

[10] EXT4 File System. `https://ext4.wiki.kernel.org/index.php/Main_Page`, 2016.

[11] Facebook RocksDB. `http://rocksdb.org/`, 2016.

[12] Flexible I/O Tester. `https://github.com/axboe/fio`, 2016.

[13] A. P. Foong, B. Veal, and F. T. Hady. Towards SSD-Ready Enterprise Platforms. In *VLDB*, 2010.

[14] Graph500. `http://www.graph500.org/`.

[15] C. Jonathan. JLS: Increasing VFS Scalability. `https://lwn.net/Articles/360199/`. 2009.

[16] C. Jonathan. Dcache scalability and RCU-walk. `https://lwn.net/Articles/419811/`. 2010.

[17] J. Kang, B. Zhang, T. Wo, W. Yu, L. Du, S. Ma, and J. Huai. SpanFS: A Scalable File System on Fast Storage Devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 249–261, Santa Clara, CA, 2015. USENIX Association.

[18] H.-J. Kim, Y.-S. Lee, and J.-S. Kim. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, June 2016. USENIX Association.

[19] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *8th USENIX Conference on File and Storage Technologies (FAST 10)*. USENIX Association, Feb. 2010.

[20] P. Kumar and H. H. Huang. G-Store: High-Performance Graph Store for Trillion-Edge Processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[21] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-Scale Graph Computation on Just a PC. In *OSDI*, 2012.

[22] W. Li, G. Jean-Baptise, J. Riveros, G. Narasimhan, T. Zhang, and M. Zhao. CacheDedup: In-line Deduplication for Flash Caching. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 301–314, Santa Clara, CA, Feb. 2016. USENIX Association.

[23] H. Liu and H. H. Huang. Graphene: Fine-Grained IO Management for Graph Computing. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[24] W. Long. [PATCH] dcache: Translating dentry into pathname without taking rename_lock. `https://lkml.org/lkml/2013/9/4/471`. 2013.

[25] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, Feb. 2016. USENIX Association.

[26] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, 2017.

[27] S. Mandal, G. Kuenning, D. Ok, V. Shastry, P. Shilane, S. Zhen, V. Tarasov, and E. Zadok. Using Hints to Improve Inline Block-layer Deduplication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 315–322, Santa Clara, CA, Feb. 2016. USENIX Association.

[28] C. Min, S. Kashyap, S. Maass, and T. Kim. Understanding Many-core Scalability of File Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, Denver, CO, 2016. USENIX Association.

[29] A. Montresor, F. De Pellegrini, and D. Miorandi. Distributed k-Core Decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 2013.

[30] J. Oh, W.-S. Han, H. Yu, and X. Jiang. Fast and Robust Parallel SGD Matrix Factorization. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 865–874. ACM, 2015.

[31] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[32] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 2013.*

[33] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment*, 6(6):433–444, 2013.

[34] E. Seppanen, M. T. O'Keefe, and D. J. Lilja. High Performance Solid State Storage Under Linux. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, May 2010.

[35] Y. Shiloach and U. Vishkin. An O(log n) Parallel Connectivity Algorithm. *Journal of Algorithms*, 1982.

[36] D. I. Shin, Y. J. Yu, H. S. Kim, J. W. Choi, D. Y. Jung, and H. Y. Yeom. Dynamic Interval Polling and Pipelined Post I/O Processing for Low-Latency Storage Class Memory. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, San Jose, CA, 2013. USENIX.

[37] W. Shin, Q. Chen, M. Oh, H. Eom, and H. Y. Yeom. OS I/O Path Optimizations for Flash Solid-state Drives. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 483–488, Berkeley, CA, USA, 2014. USENIX Association.

[38] SNIA. `http://nvmexpress.org/wp-content/uploads/2013/04/NVM_whitepaper.pdf`,.

[39] Specification for Supermicro Product(X9DRG-HF). `http://www.supermicro.com/products/motherboard/Xeon/C600/X9DRG-HF.cfm`, 2016.

[40] I. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska. sRoute: Treating the Storage Stack Like a Network. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 197–212, Berkeley, CA, USA, 2016. USENIX Association.

[41] UMass Trace Repository. `http://traces.cs.umass.edu/index.php/Storage/Storage`, 2012.

[42] K. Vora, G. Xu, and R. Gupta. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 507–522, Denver, CO, 2016. USENIX Association.

[43] D. Vučinić, Q. Wang, C. Guyot, R. Mateescu, F. Blagojević, L. Franca-Neto, D. L. Moal, T. Bunker, J. Xu, S. Swanson, and Z. Bandić. DC Express: Shortest Latency Protocol for Reading Phase Change Memory over PCI Express. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 309–315, Santa Clara, CA, 2014. USENIX.

[44] H. Wang and P. Varman. Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-Aware Allocation. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 229–242, Santa Clara, CA, 2014. USENIX.

[45] J. Yang, D. B. Minturn, and F. Hady. When Poll is Better Than Interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

[46] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, J. W. Choi, H. S. Kim, H. Eom, and H. Y. Yeom. Optimizing the Block I/O Subsystem for Fast Storage Devices. *ACM Transaction on Computer System (TOCS)*, 32(2):6:1–6:48, June 2014.

[47] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, H. Eom, and H. Y. Yeom. Exploiting Peak Device Throughput from Random Access Workload. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[48] J. Zhang, J. Shu, and Y. Lu. ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 87–100, Denver, CO, 2016. USENIX Association.

[49] D. Zheng, R. Burns, and A. S. Szalay. Toward Millions of File System IOPS on Low-cost, Commodity Hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 69:1–69:12, New York, NY, USA, 2013. ACM.

[50] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[51] X. Zhu, W. Han, and W. Chen. GridGraph: Large-scale Graph Processing on a Single Machine Using 2-level Hierarchical Partitioning. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference*, 2015.