Different I/O Access Methods for Linux, What We Chose for Scylla, and Why

By Avi Kivity
October 5, 2017

From original publication: https://www.scylladb.com/2017/10/05/io-access-methods-scylla/
Figures adapted to a more printable and viewable form paper by Sam Siewert for academic use.

When most server application developers think of I/O, they consider network I/O since most resources these days are accessed over the network: databases, object storage, and other microservices. The developer of a database, however, also has to consider file I/O. This article describes the available choices and their tradeoffs and why Scylla chose asynchronous direct I/O (AIO/DIO) as its access method.

Choices for accessing files

In general, there are four choices for accessing files on a Linux server: read/write, mmap, Direct I/O (DIO) read/write, and asynchronous direct I/O (AIO/DIO).

Traditional read/write

The traditional method, available since the beginning of time, is to use the *read(2)* and *write(2)* system calls. In a modern implementation, the read system call (or one of its many variants – *pread*, *readv*, *preadv*, etc) asks the kernel to read a section of a file and copy the data into the calling process address space. If all of the requested data is in the page cache, the kernel will copy it and return immediately; otherwise, it will arrange for the disk to read the requested data into the page cache, block the calling thread, and when the data is available, it will resume the thread and copy the data. A write, on the other hand, will usually[1] just copy the data into the page cache; the kernel will write-back the page cache to disk some time afterward. See Figure 1.

Mmap

An alternative and more modern method is to memory-map the file into the application address space using the *mmap(2)* system call. This causes a section of the address space to refer directly to the page cache pages that contain the file's data. After this preparatory step, the application can access file data using the processor's memory read and memory write instructions. If the requested data happens to be in cache, the kernel is completely bypassed and the read (or write) is performed at memory speed. If a cache miss occurs, then a page-fault happens and the kernel puts the active thread to sleep while it goes off to read the data for that page. When the data is finally available, the memory-management unit is programmed so the newly read data is accessible to the thread which is then woken. See Figure 2.

Figure 1 – Traditional read/write

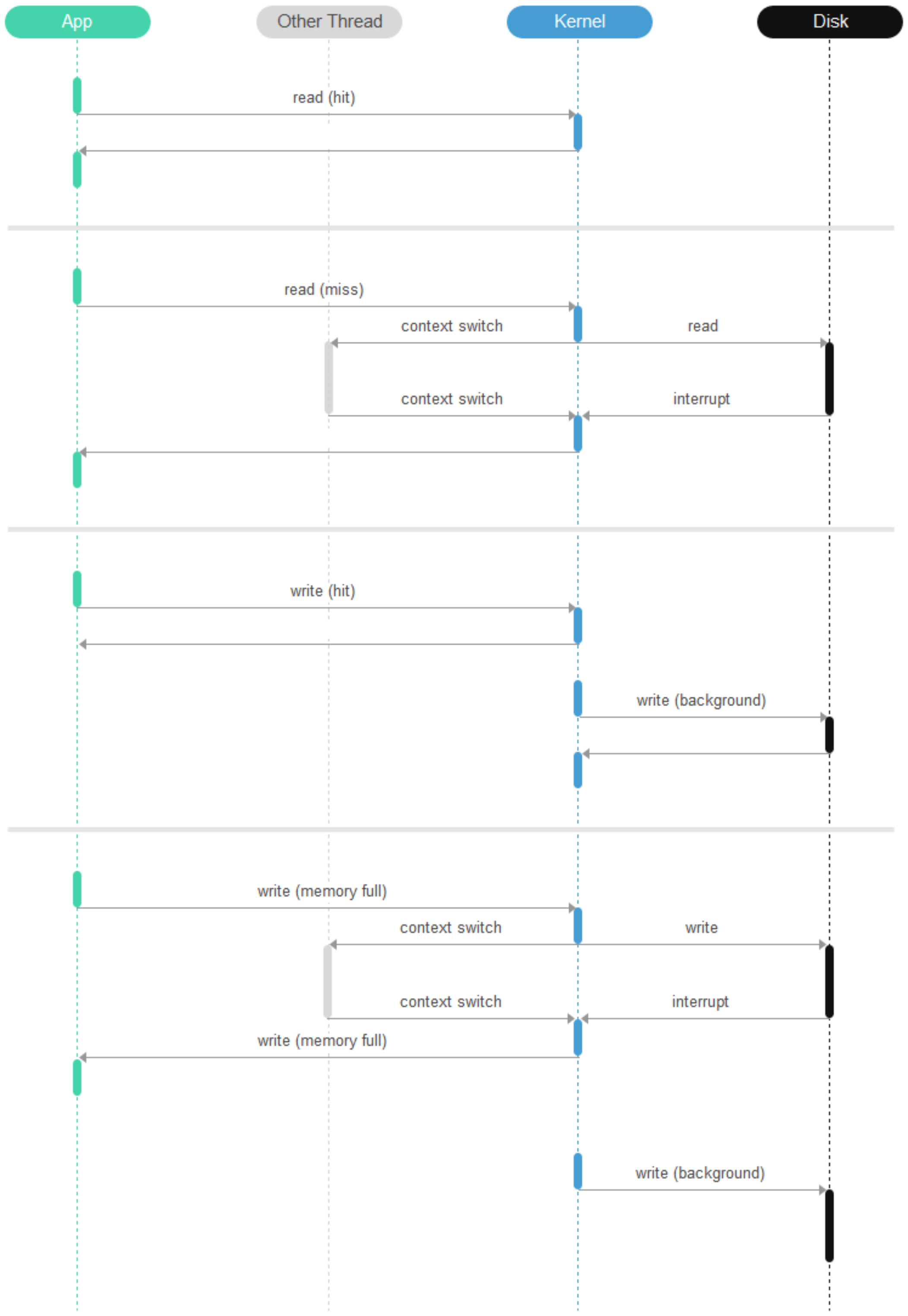
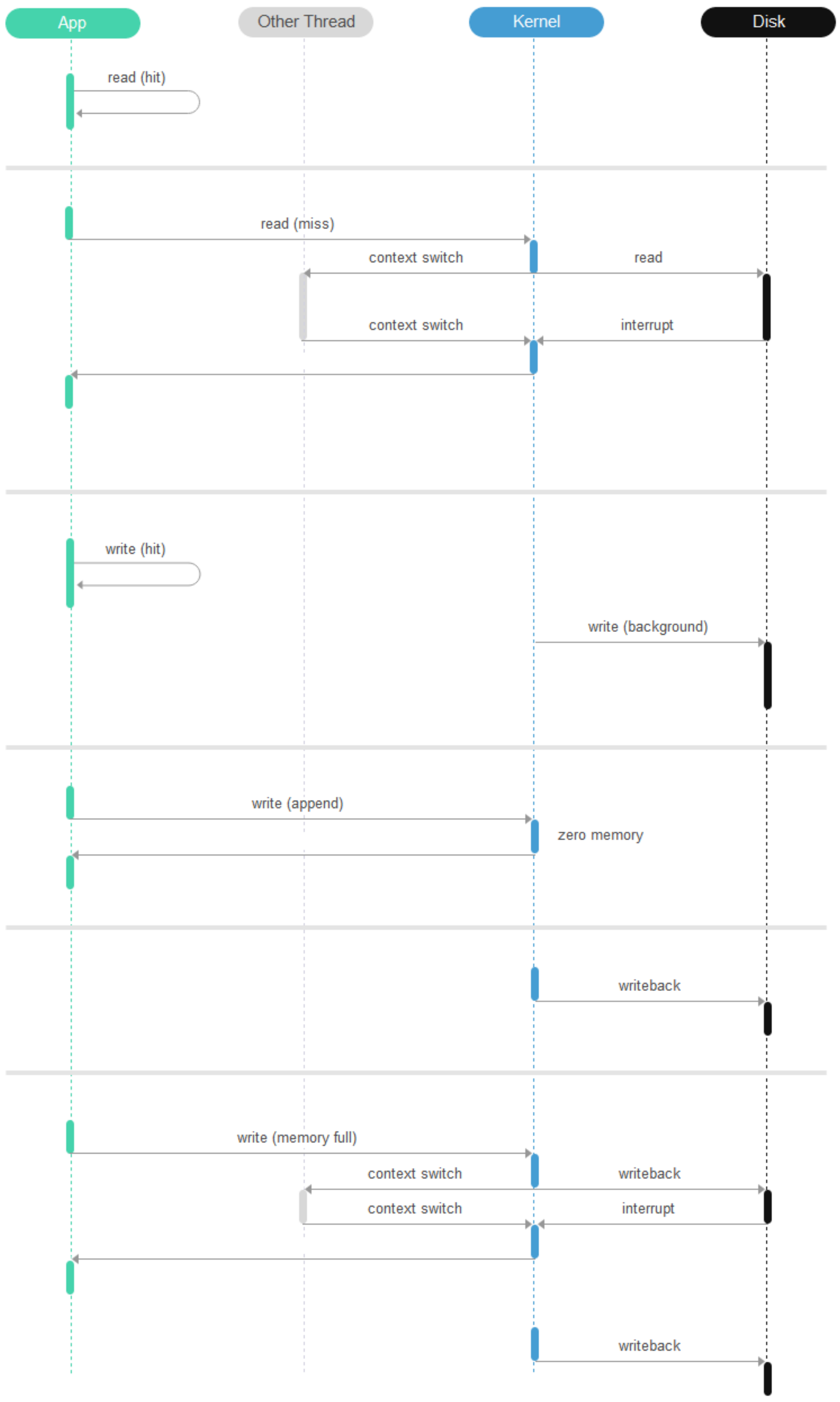


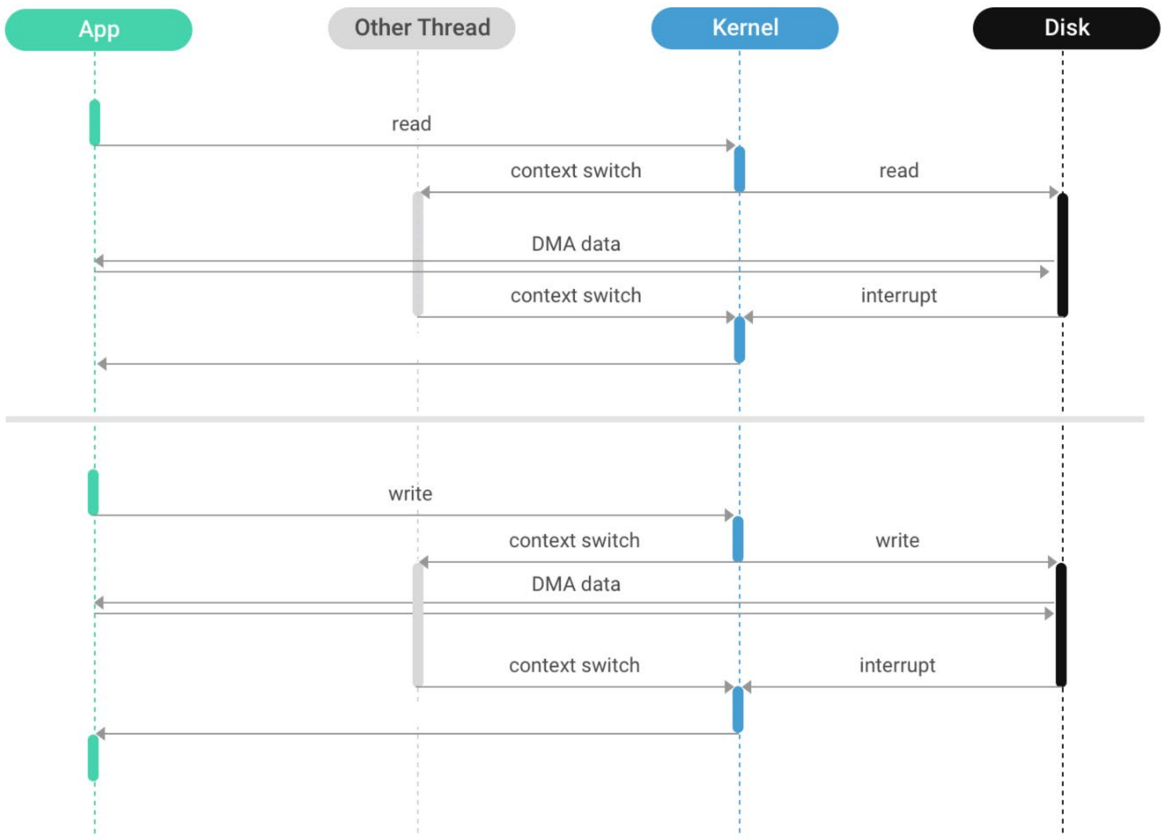
Figure 2 – MMAP



Direct I/O (DIO)

Both traditional read/write and mmap involve the kernel page cache and defer scheduling of I/O to the kernel. When the application wishes to schedule I/O itself (for reasons that we will explain later), it can use direct I/O. This involves opening the file with the *O_DIRECT* flag; further activity will use the normal read and write family of system calls, but their behavior is now altered: instead of accessing the cache, the disk is accessed directly which means that the calling thread will be put to sleep unconditionally. Furthermore, the disk controller will copy the data directly to userspace, bypassing the kernel. See Figure 3.

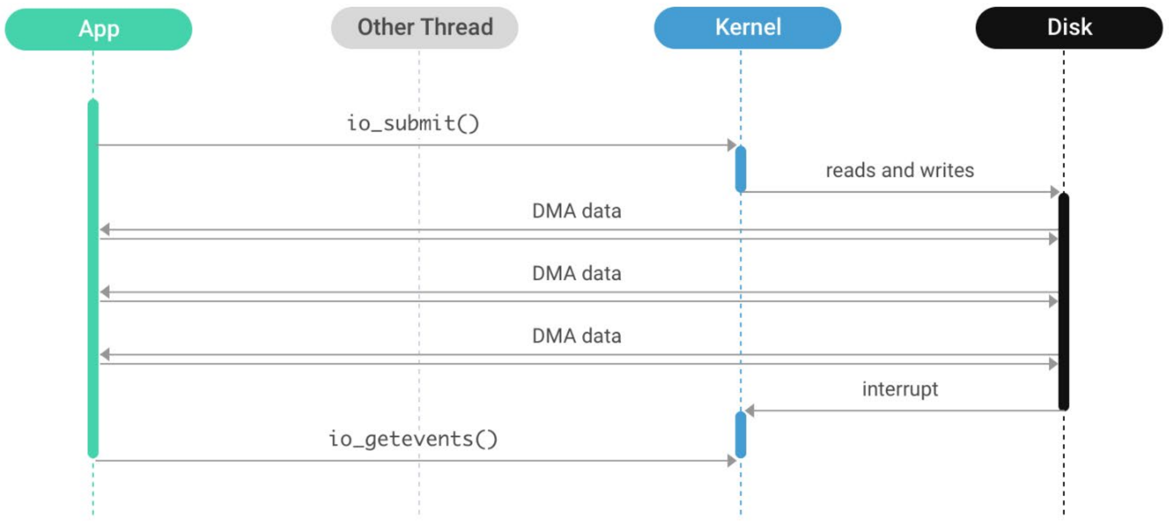
Figure 3 – Direct I/O



Asynchronous direct I/O (AIO/DIO)

A refinement of direct I/O, asynchronous direct I/O behaves similarly but prevents the calling thread from blocking. Instead, the application thread schedules direct I/O operations using the *io_submit(2)* system call, but the thread is not blocked; the I/O operation runs in parallel with normal thread execution. A separate system call, *io_getevents(2)*, is used to wait for and collect the results of completed I/O operations. Like DIO, the kernel's page cache is bypassed, and the disk controller is responsible for copying the data directly to userspace. See Figure 4.

Figure 4 – Asynchronous Direct I/O



Understanding the tradeoffs

The different access methods share some characteristics and differ in some others. Table 1 summarizes these characteristics, which are elaborated on below.

Characteristic	R/W	mmap	DIO	AIO/DIO
Cache control	kernel	kernel	user	user
Copying	yes	no	no	no
MMU activity	low	high	none	none
I/O scheduling	kernel	kernel	mixed	user
Thread scheduling	kernel	kernel	kernel	user
I/O alignment	automatic	automatic	manual	manual
Application complexity	low	low	moderate	high

Cache control

For both read/write and mmap, caching is the responsibility of the kernel. The majority of the system's memory is given over to the page cache. The kernel decides which pages should be evicted when memory runs low; when pages need to be written back to disk; and controls read-ahead. The application can provide some guidance to the kernel using the *madvise(2)* and *fadvise(2)* system calls.

The great advantage of letting the kernel control caching is that great effort has been invested by the kernel developers over many decades into tuning the algorithms used by the cache. Those algorithms are used by thousands of different applications and are generally effective. The disadvantage, however, is that these algorithms are general-purpose and not tuned to the application. The kernel must guess how the application will behave next, and even if the application knows differently, it usually has no way to help the kernel guess correctly. This results in the wrong pages being evicted, I/O scheduled in the wrong order, or read-ahead scheduled for data that will not be consumed in the near future.

Copying and MMU activity

One of the benefits of the mmap method is that if the data is in cache then the kernel is bypassed completely. The kernel does not need to copy data from the kernel to userspace and back, so fewer processor cycles are spent on that activity. This benefits loads that are mostly in cache (for example, if the ratio of storage size to RAM size is close to 1:1).

The downside of mmap, however, occurs when data is not in the cache. This usually happens when the ratio of storage size to RAM size is significantly higher than 1:1. Every page that is brought into cache causes another page to be evicted. Those pages have to be inserted into and removed from the page tables; the kernel has to scan the page tables to isolate inactive pages, making them candidates for eviction, and so forth. In addition, mmap requires memory for the page tables. On x86 processors, this requires 0.2% of the size of the mapped files. This seems low, but if the application has a 100:1 ratio of storage to memory, the result is that 20% of memory (0.2% * 100) is devoted to page tables.

I/O scheduling

One of the problems with letting the kernel control caching (with the mmap and read/write access methods) is that the application loses control of I/O scheduling. The kernel picks whichever block of data it deems appropriate and schedules it for write or read. This can result in the following problems:

- A write storm: when the kernel schedules large amounts of writes, the disk will be busy for a long while and impact read latency
- The kernel cannot distinguish between "important" and "unimportant" I/O. I/O belonging to background tasks can overwhelm foreground tasks which will impact their latency[2]

By bypassing the kernel page cache, the application takes the burden of scheduling I/O on itself. This doesn't mean that the problems are solved; but it does mean that the problems can be solved, with sufficient attention and effort.

When using Direct I/O, each thread controls when to issue I/O However, the kernel controls when the thread runs so responsibility for issuing I/O is shared between the kernel and the application. With AIO/DIO, the application is in full control of when I/O is issued.

Thread scheduling

An I/O intensive application using mmap or read/write cannot guess what its cache hit rate will be. Therefore it has to run a large number of threads (significantly larger than the core count of the machine it is running on). Using too few threads, they may all be waiting for the disk leaving the processor underutilized. Since each thread usually has at most one disk I/O outstanding, the number of running threads must be around the concurrency of the storage subsystem multiplied by some small factor in order to keep the disk fully occupied. However, if the cache hit rate is sufficiently high, then these large number of threads will contend with each other for the limited number of cores.

When using direct I/O, this problem is somewhat mitigated as the application knows exactly when a thread is blocked on I/O and when it can run, so the application can adjust the number of running threads according to runtime conditions.

With AIO/DIO, the application has full control over both running threads and waiting I/O (the two are completely divorced); so it can easily adjust to in-memory or disk-bound conditions or anything in between.

I/O alignment

Storage devices have a block size; all I/O must be performed in multiples of this block size which is typically 512 or 4096 bytes. Using read/write or mmap, the kernel performs the alignment automatically; a small read or write is expanded to the correct block boundary by the kernel before it is issued.

With DIO, it is up to the application for perform block alignment. This incurs some complexity but also provides an advantage: the kernel will usually over-align to a 4096 byte boundary even when a 512-byte boundary suffices but a user application using DIO can issue 512-byte aligned reads which results in saving bandwidth on small items.

Application complexity

While the previous discussions favored AIO/DIO for I/O intensive applications, that method comes with a significant cost: complexity. Placing the responsibility of cache management on the application means it can make better choices than the kernel and make those choices with less overhead. However, those algorithms need to be written and tested. Using asynchronous I/O requires that the application is written using callbacks, coroutines, or a similar method, and often reduces the reusability of many available libraries.

Scylla and AIO/DIO

With Scylla, we have chosen the highest performing option, AIO/DIO. To isolate some of the complexity involved, we wrote Seastar, a high-performance framework for I/O intensive applications. Seastar abstracts away the details of performing AIO and provides common APIs for network, disk, and multi-core communications. It also provides both callback and coroutine styles of state management suitable for different use cases.

Different areas of Scylla highlight different ways that I/O can be used:

- Compaction uses application-level read-ahead and write-behind to ensure high throughout but bypass application level caches due to expected low hit rates (and to avoid flooding the cache with cold data)
- Queries (reads) use application-controlled read-ahead and application-level caching. Application-controlled read-ahead prevents read-ahead over-runs since we know the boundaries of the data on disk ahead of time. Also application-level caching allows us to cache not only the data read from disk but also the work that went into merging data from multiple files into a single cache item
- Small reads are aligned to a 512-byte boundary to reduce bus data transfers and latency
- The Seastar I/O scheduler allows us to dynamically control I/O rates for compaction and queries (as well as other operation classes) to satisfy user service-level agreements (SLAs)
- A separate I/O scheduling class ensures that commitlog writes get the required bandwidth and are not dominated by reads or dominate reads

AIO/DIO is a good starting point for directly driving NVMe drives from the application to further bypass the kernel. This may become a future Seastar feature.

Conclusions

We’ve shown four different ways to perform disk I/O on Linux and the different tradeoffs involved in them. It is easy to start with the traditional read/write operations or get good in-memory performance with mmap, but to achieve top performance and control, we’ve chosen asynchronous I/O for Scylla.

[1.] The exceptions are a write that is not aligned on a page boundary that modifies an uncached page; and a write that needs to allocate a new cache page, when memory is not available.

[2.] There is limited ability to communicate I/O priority to the kernel with the *ioprio_set(2)* system call
deep-divedevelopmentinternals

About Avi Kivity

Avi Kivity, CTO of ScyllaDB, is known mostly for starting the Kernel-based Virtual Machine (KVM) project, the hypervisor underlying many production clouds. He has worked for Qumranet and Red Hat as KVM maintainer until December 2012. Avi is now CTO of ScyllaDB, bringing high throughput to the NoSQL world.