WHITE PAPER

Data Center
Storage

Achieving Optimal Performance & Endurance on Coarse Indirection Unit SSDs

Recent increases in storage density allow larger capacity SSDs to be built, utilizing bigger IU sizes. Here are techniques on how to efficiently use coarse IU sized SSDs.

Table of Contents

Authors

Andrzej Jakowski,
NSG Software Architect

Keith Busch,
SSG/OTC Software Architect

Michael Reed,
NSG Strategic Planner

Michael Allison,
NSG Principal Engineer

Purpose

Purpose

This white paper provides best known methods and practices on how to efficeintly use high capacity SSDs that utilize greater than 4KiB Indirection Unit (IU). This includes, but is not limited to, some of the latest Intel® QLC 3D NAND SSDs.

Scope

This white paper focuses on the Linux* operating system (OS) utilizing traditional I/O paths. Introducing the read-modify-write (RMW) cycle concept and its system level implications. Linux I/O stacks are illustrated and different types of I/O are explained. Finally, we explore BKMs and practices to maximize performance and endurance for coarse IU SSDs.

Target Audience

Targeted for application developers, system administrators, and system operators wanting to configure underlying software and hardware storage infrastructure. This document assumes familiarity with basic software development and storage related terminology.

Glossary

- *Indirection Unit (IU):* An object of a particular size that can be accessed using some kind of reference; e.g. name, id, pointer, etc. For example, logical block (sector) represents IU for the storage devices. Actual user data on the storage device may be stored in different physical locations on the media, but users always reference that data by providing the logical block address (LBA). SSD IU refers to the internal construct that the SSD uses to manage data placement on the physical media.

- *Read-Modify-Write:* A process of updating content in memory or storage. It consists of reading old content, merging it with new content, and subsequently writing the updated data back to the media.

Background Information

Standard capacity SSDs, which utilize 4KiB IUs, work well with existing host software. This is because a majority of writes submitted by host software to the SSD are in page (4KiB for x86 based systems) granularity and alignment. It is possible for the host system to submit sub-IU-sized or misaligned writes to the SSD (e.g. 512B), but these writes are not optimal from a performance and endurance perspective.

Recent increases in storage density allow larger capacity SSDs to be built. Such SSDs will utilize bigger IU sizes (e.g. 16KiB) to reduce SSD cost. Customers who want to efficiently use coarse IU-sized SSDs should modify their software so that writes issued are IU aligned and multiple in size of IU. It is still possible for the host system to issue writes that are smaller than IU (e.g. 4KiB), but SSD performance and endurance will be impacted.

SSD Endurance Management and Read-Modify Cycle

To maximize its endurance, modern SSDs apply wear leveling techniques to determine best data placement on the media. One of the wear leveling techniques is to map logical blocks (LB) accessible by the host system to the physical locations on the media, so the same LB can point to different physical locations inside the SSD. The SSD controller manages that map and determines where to place user data on the media. This allows all the media blocks to be written approximately the same number of times, thus maximizing SSD endurance.

LB-to-physical location map typically associates 512B LB to bigger-sized units, for example 4KiB IUs. This technique enables SSD cost reduction associated with storing the indirection map, but it may negatively impact SSD write performance and endurance.

Figure 1 shows an example storage device implementing this concept. The example storage device is divided into smaller units called sectors. A sector is the minimal unit that can be read from, or written to, and is typically 512B in size. Several sectors (8 in the given example) are tracked together internally by the storage device controller which can only access the underlying media in the larger units (4KiB in size in this example). The image illustrates optimal and non-optimal writes to that storage device. Optimal write has the following characteristics:

1. **It is correctly sized** – its size is equal to multiple of "Internal storage device unit," e.g. 4KiB.
2. **It is correctly aligned** – its starting address is aligned to multiple of "Internal storage device unit," e.g. write starts at the 8th sector.

Figure 1 also illustrates writes that are not optimal from the storage device perspective. The first red-colored write is misaligned – it starts at the 25th sector. This write requires the storage device to read old data from two units (shown in grey), merge it with new data, and write two units back to the media. A similar situation happens for the second red-colored write operation, for which I/O is properly aligned but its size is smaller than the unit size. This is why the storage device controller needs to read the whole unit, merge it with new data, and write the single unit to the media. Actual amount of data written to the media is bigger in size than original user-initiated write.

Figure 1 illustrates the concept of RMW cycle and its negative impact on:

1. **Performance**: A single write operation in RMW cycle requires storage device to perform two I/Os: read the old content from the media, merge it with new content, and write all the data pieces back to the underlying media
2. **SSD endurance**: In the RMW cycle, the storage device has to write more data than it was originally requested to write. As shown by the first red-colored write in the given example, the storage device was requested to store eight sectors. This write triggered the RMW cycle, which ultimately caused 16 sectors to be written back to the underlying media.

As indicated earlier, for certain high capacity Intel SSDs SKUs, IU size will be increased. While different IU sizes may be offered (e.g. 16KiB, 64KiB) depending on the SSD SKU, this whitepaper provides BKMs and practices that are applicable to any IU size.

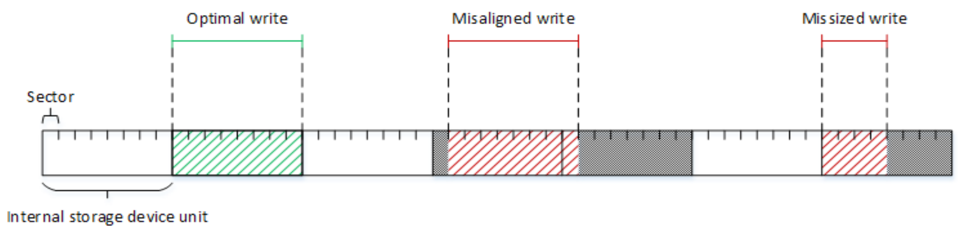


Figure 1. Example Storage Device Layout

Linux I/O Stack

This section illustrates Linux I/O stack software layers, and explains different types of I/O. This section is an introduction to the applicable techniques for optimizing I/O for coarse IU SSDs.

File System and Block Layer SW Stack

Figure 2 illustrates a simplified model for the file system and block layer software stack in the Linux kernel. The I/O path starts at the application layer. Typically applications (e.g. database management systems) issue I/O to the kernel through system libraries or directly through the system call interface. Application I/O is intercepted by a virtual file system (VFS) which provides a common interface for different file system types.

The path from VFS directly to the block layer is called raw I/O (black arrows in the diagram), whereas the path via logical file system (e.g. XFS) is called file system I/O (blue arrows in the diagram). When I/O hits the block layer, it typically goes to the low level disk driver and then to a physical storage device. Alternatively, it is submitted to logical volume devices' infrastructure (e.g. RAID, LVM), which redirects the I/O to the proper physical device.

The Linux kernel provides a number of different types of cache which are designed to improve the performance of file system I/O. The page cache, shown in gray, caches virtual memory pages, including file system pages. The size of the page cache is dynamic and increases to use available memory. When the kernel determines that page cache memory is needed for other purposes (e.g. to run user space programs), it will decrease page cache size.

By default, file systems interact with page cache transparently from the application perspective; caching data to improve read performance, and buffering data to improve write performance. When an application developer wants to avoid file system caching, he or she can issue direct I/O which instructs VFS and the logical file system to omit page cache.

I/O Types

Applications issue different types of I/O to achieve different desired goals (e.g. to make sure that data was written to persistent media, etc.). Below are simplified explanations of different I/O classification types:

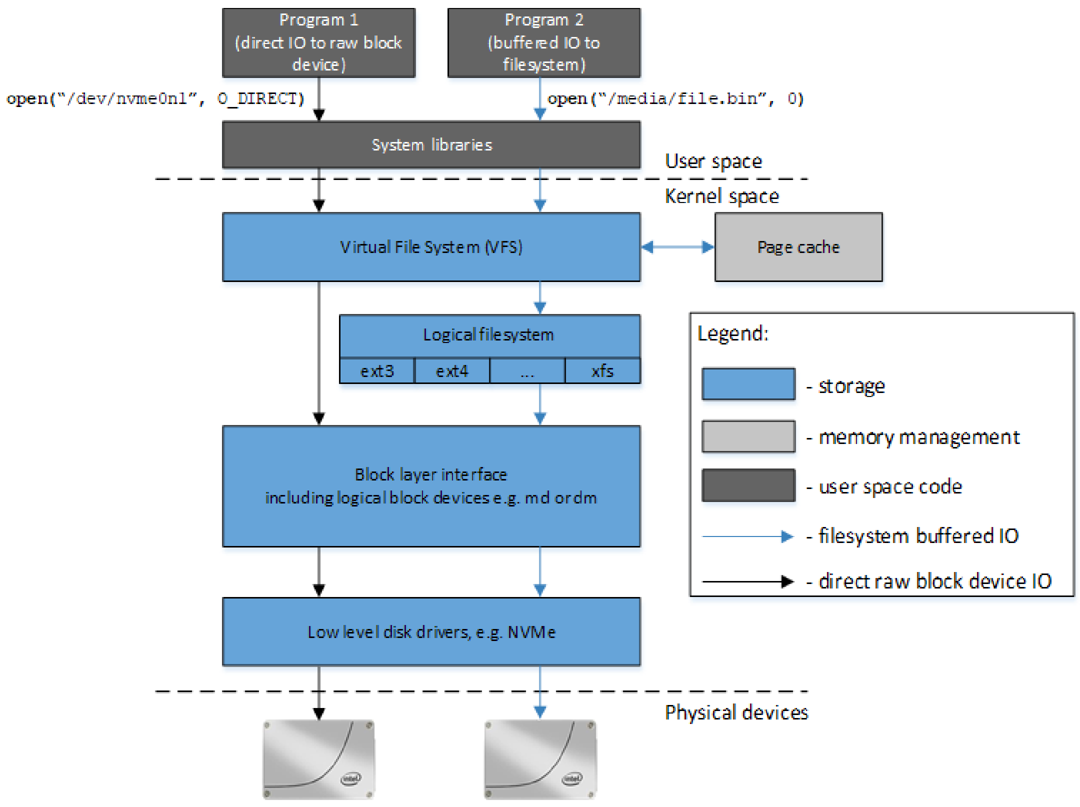

Program 1
(direct IO to raw block device)
Program 2
(buffered IO to filesystem)
open("/dev/nvme0n1", O_DIRECT)
open("/media/file.bin", 0)
System libraries
User space
Kernel space
Virtual File System (VFS)
Page cache
Logical filesystem
ext3
ext4
...
xfs
Block layer interface
including logical block devices e.g. md or dm
Low level disk drivers, e.g. NVMe
Physical devices
Legend:
- storage
- memory management
- user space code
- filesystem buffered IO
- direct raw block device IO

Figure 2. Linux I/O Stack

- **Filesystem I/O** – Issued to the file in the logical file system.
- **Raw I/O** – Issued directly to the disk, bypassing logical file system.
- **Direct I/O** – Allows applications to issue I/O but bypass page cache. This can be useful for applications that maintain their own caches and want to avoid the effect of double caching, or do not want to pollute page cache (e.g. in case of backup operation). The direct I/O can be issued as raw I/O directly to the disk, or as file system I/O – to the file in the file system.

  An application developer wanting to issue direct I/O needs to specify `O_DIRECT` flag to `open()` system call and prepare memory buffers for transferring the data.

  Original filesystem direct I/O may be modified by the filesystem to match the requirements of that filesystem, e.g. it can be mapped to the actual disk location or resized to match the filesystem block size.
- **Non-blocking I/O** – When an application issues I/O, it will complete immediately (i.e. when data is retrieved from page cache) or the application will blocked until the data becomes available. Non-blocking I/O, if supported by the filesystem, allows the application to continue after issuing I/O, without waiting for its completion.

  To use the non-blocking I/O interface, application developers must specify `O_NONBLOCK` or `O_NDELAY` flag to `open()`system call. Actual read or write operation will return an error indicating to the application that the I/O has not completed, or will report success when the I/O has completed.
- **Synchronous I/O** – When specified during opening a file (`O_SYNC` flag supplied to `open()` syscall), the kernel will make sure that write I/O will write the data, and all the associated metadata, to the underlying storage device.

Techniques and Considerations Allowing to Optimize I/O for Coarse IU SSDs

This section explains the architectural choices made while implementing existing filesystem software stacks in the Linux kernel and coarse IU SSDs; choices limiting optimal use of both in all the use cases.

It also explains choices that allow application developers to optimize writes to coarse IU SSDs. Finally, it provides example techniques, BKMs, and code snippets showing how to implement these techniques.

File System Software Stack and High Capacity SSD Architectural Choices

Current VFS and logical filesystem software stacks are tightly integrated with page cache and treat the memory page as a fundamental unit while accessing underlying disks. In x86 based systems, page size is equal to 4,096B and a significant number of disk accesses are performed in page granularities. One of the consequences of the tight integration between page cache, VFS, and logical filesystems, such as XFS and ext4, is the maximum block size for those filesystem is page size. This may effectively cause one big application I/O to be translated by a filesystem into a number of small, page-sized I/Os.

On the other hand, some high capacity Intel® SSD SKUs will utilize bigger than 4KiB IUs - s large as 16KiB. Thus, any write to the SSD that is smaller than, or not aligned to the IU (16KiB in the example) may introduce RMW cycle in the SSD. As indicated earlier, RMW may negatively impact SSD performance and endurance. Because of this, sub-IU writes are not optimal from the SSD perspective.

Recommended Techniques and Considerations for Optimizing I/O

To achieve optimal performance and endurance on coarse IU SSDs, consider the following techniques:

1. Properly size and align partitions to the IU, or its multiple, as illustrated in Figure 3. The same technique may apply to any software or hardware solution implementing logical volume capability on coarse IU SSDs. For example, it may include selecting proper stripe size for RAID. Please refer to the documentation of the specific product.

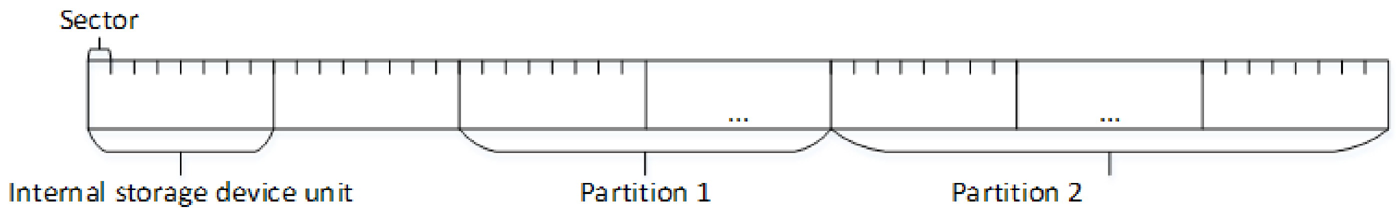
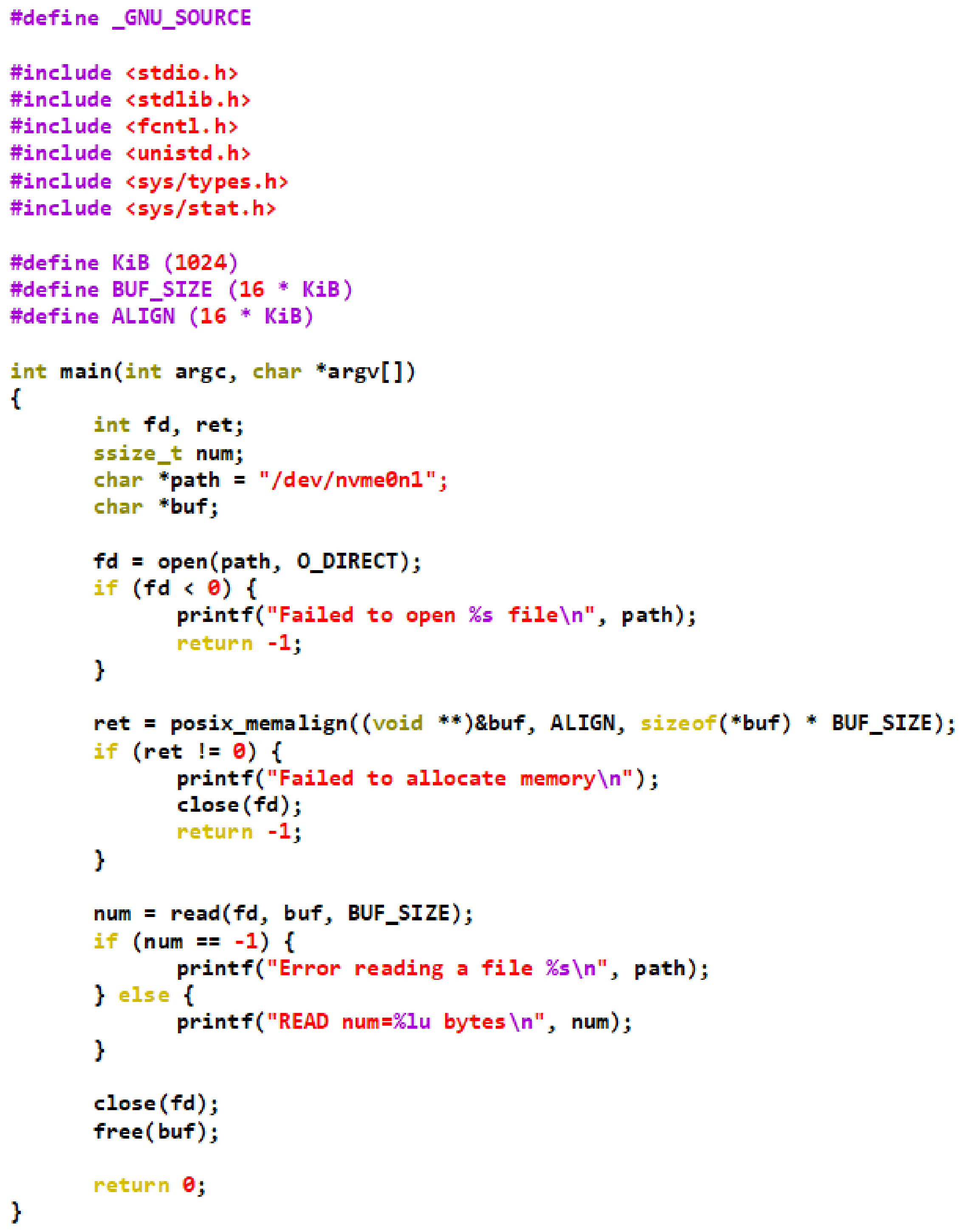


Figure 3. Optimal SSD Partition Alignment

2. To achieve optimal I/O performance and endurance on coarse IU SSDs, perform I/O directly on the raw block device file (e.g. /dev/nvme0n1) bypassing the logical file system and page cache (open file with `O_DIRECT` flag). For optimal performance, I/O should be sized and aligned to IU or its multiple. Please refer to Figure 4 for example code snippet that can be incorporated to the software.



```c
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

#define KiB (1024)
#define BUF_SIZE (16 * KiB)
#define ALIGN (16 * KiB)

int main(int argc, char *argv[])
{
        int fd, ret;
        ssize_t num;
        char *path = "/dev/nvme0n1";
        char *buf;

        fd = open(path, O_DIRECT);
        if (fd < 0) {
                printf("Failed to open %s file\n", path);
                return -1;
        }

        ret = posix_memalign((void **)&buf, ALIGN, sizeof(*buf) * BUF_SIZE);
        if (ret != 0) {
                printf("Failed to allocate memory\n");
                close(fd);
                return -1;
        }

        num = read(fd, buf, BUF_SIZE);
        if (num == -1) {
                printf("Error reading a file %s\n", path);
        } else {
                printf("READ num=%lu bytes\n", num);
        }

        close(fd);
        free(buf);

        return 0;
}
```

Figure 4. Code snippet showing example implementation of optimal I/O: direct I/O performed on raw block device that adheres to IU size and alignment recommendations (in this example IU is 16KiB in size)

Future work and final notes

Potential future Intel work includes:

- Ecosystem enablement, including collaboration with standardization bodies, open source communities, and OSVs to define mechanism for discovery I/O optimization attributes, such as recommended write size and alignment. This will allow application developers to modify their software infrastructure to utilize those attributes in their software and thus achieve optimal performance and endurance from coarse IU SSDs.
- Documenting the optimal performance and endurance techniques through whitepapers/how-to document, etc.