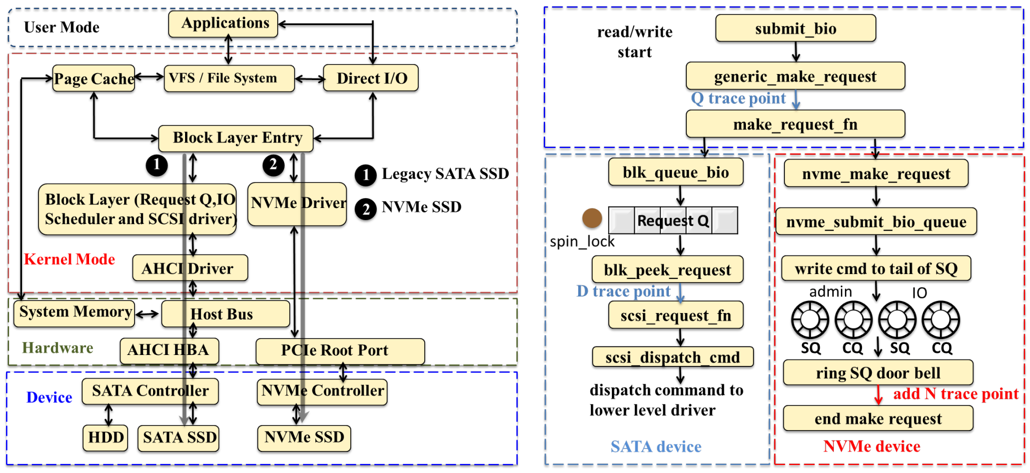

(a) (b)

Figure 1: (a) A comparison of the software and hardware architecture of SATA HDD, SATA SSD, and NVMe SSD. (b) An illustration of the I/O software stack of SATA SSD and NVMe SSD.

Hardware data path: Figure 1(a) depicts the data access paths of both SATA and NVMe SSDs. A typical SATA device connects to the system through the host bus. An I/O request in such a device traverses the AHCI driver, the host bus, and an AHCI host bus adapter (HBA) before getting to the device itself. On the other hand, NVMe SSDs connect to the system directly through the PCIe root complex port. In Section 3 we show that NVMe's shorter data path significantly reduces the overall data access latency.

Simplified software stack: Figure 1(b) shows the software stack for both NVMe and SATA based devices. In the conventional SATA I/O path, an I/O request arriving at the block layer will first be inserted into a request queue (*Elevator*). The Elevator would then reorder and combine multiple requests into sequential requests. While reordering was needed in HDDs because of their slow random access characteristics, it became redundant in SSDs where random access latencies are almost the same as sequential. Indeed, the most commonly used Elevator scheduler for SSDs is the `noop` scheduler (Rice 2013), which implements a simple First-In-First-Out (FIFO) policy without any reordering. As we show in Section 3, the Elevator being a single point of contention, significantly increases the overall access latency.

The NVMe standard introduces many changes to the aforementioned software stack. An NVMe request bypasses the conventional block layer request queue[2]. Instead, the NVMe standard implements a paired *Submission* and *Completion* queue mechanism. The standard supports up to 64 K I/O queues and up to 64 K commands per queue. These queues are saved in host memory and are managed by the NVMe driver and the NVMe controller cooperatively: new commands are placed by the host software into the submission queue (SQ), and completions are placed into an associated completion queue (CQ) by the controller (Huffman 2012). The controller fetches commands from the front of the queue, processes the commands, and rings the completion queue doorbell to notify the host. This asynchronous handshake reduces CPU time spent on synchronisation, and removes the single point of contention present in legacy block drivers.

[2] The 3.17 linux kernel has a re-architected block layer for NVMe drives based on (Bjørling et al. 2013). NVMe accesses no longer bypass the kernel. The results shown in this paper are for the 3.14 kernel.

3. NVMe Performance Analysis

In this section we provide detailed analysis of NVMe performance in comparison to SATA-based HDDs and SSDs. First, in Section 3.1, we describe our driver instrumentation methodology and the changes to `blktrace` that were done to provide access latency break downs. Sections 3.2 and Section 3.3 then use these tools to provide insight into analyzing the software stack and the device itself.

3.1 Driver and `blktrace` Instrumentation

l We use *fio* (Axboe 2014) to generate synthetic I/O traffic, and use *blktrace* (Axboe and Brunelle 2007) to characterize individual drives and their software stacks. Unless stated otherwise, I/O traffic is generated using fio's *libaio* asynchronous I/O engine. We bypass the page cache to ensure measurement of raw device performance. This is achieved by setting fio's `direct` I/O flag. For our analysis, we also use two statistics reported by fio: (1) `slat` (submission latency), which is the time taken to submit the I/O request to kernel space; and (2) `clat` (completion latency), which is the time taken by the I/O request from submission to completion.

Blktrace (Axboe and Brunelle 2007) is a tool that enables tracing the path of an I/O operation throughout the I/O stack. It relies on tracepoints inserted into the Linux kernel in order to report information about multiple trace events. This