ChatGPT

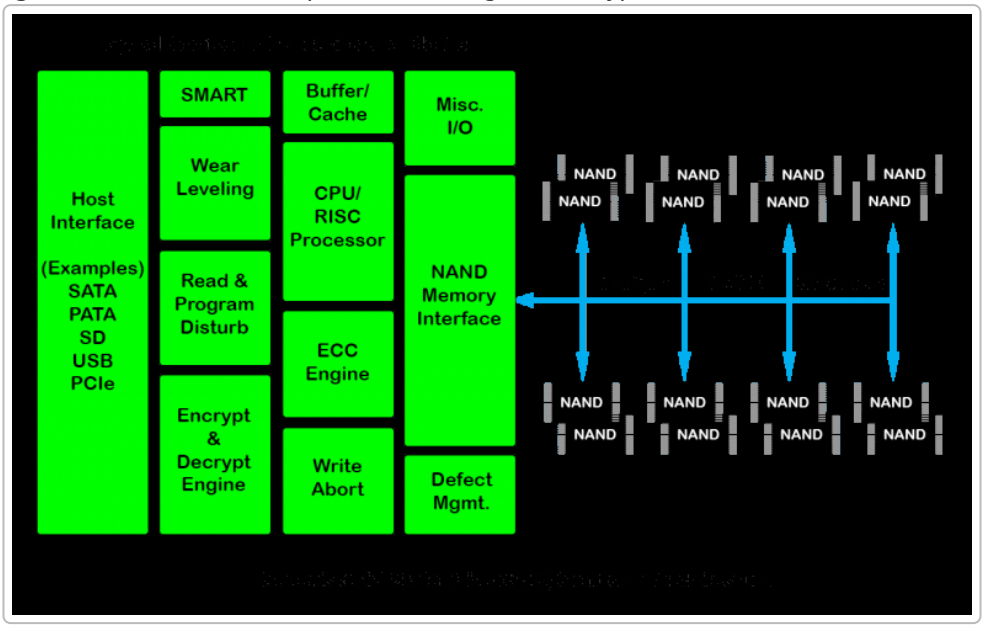
Inside Modern SSDs and NVMe Storage: Architecture, Interfaces, and Linux Integration

Introduction

Modern storage devices like **Solid-State Drives (SSDs)** have transformed data storage with their high speed and parallelism. Unlike traditional hard disks, SSDs have no moving parts – they store data on flash memory chips. To maximize performance and longevity, SSDs rely on sophisticated controllers and firmware. New interface standards such as **NVMe (Non-Volatile Memory Express)** were developed to overcome limitations of older protocols like SATA/AHCI and SCSI. This article demystifies how SSD hardware works (from flash memory characteristics to controller design and firmware functions like wear leveling and garbage collection) and how these devices communicate with computers through interfaces (AHCI, NVMe, SCSI). We also explain how SSDs and NVMe drives map into the Linux block layer (via NVMe drivers or the SCSI stack) and how their behavior influences performance, I/O scheduling, and system design decisions.

Hardware Architecture of a Modern SSD

An SSD is essentially a miniature computer dedicated to storage: it contains a **flash memory array** for data, and a specialized **SSD controller** (often with embedded processors, caches, and custom logic) that manages the flash. **Figure 1** below shows a simplified block diagram of a typical SSD's internal architecture



. On the left, the *host interface* block connects to the host computer (examples include SATA, PCIe for NVMe, etc.). Internally, the SSD controller incorporates components such as a CPU or RISC processor core, buffer/ DRAM cache for staging data, an error-correcting code (ECC) engine to detect and fix flash bit errors, and various firmware-managed functions (wear leveling, bad block management, encryption, etc.) [1] [2] . The

controller's firmware implements a **Flash Translation Layer (FTL)** which maps the host's logical block addresses to physical flash memory locations and handles housekeeping like garbage collection and wear leveling [3] [4]. On the right side of Figure 1, the controller connects to multiple **NAND flash memory chips** through one or more channels (often following standards like ONFI) [5]. Each channel can have multiple chips (sometimes called "ways"), enabling parallel operations. In essence, the SSD controller bridges the high-level host requests to low-level operations on the flash chips, managing data placement and retrieval across the many flash dies in parallel [6] [7].

Flash Memory and Parallelism: NAND flash memory, which SSDs use as storage media, has unique characteristics that influence SSD design. Flash is organized into **pages** (e.g. 4KB or larger) which are the smallest unit of read and write, and these pages are grouped into **blocks** (e.g. containing hundreds of pages, with total block sizes of a few MB) which are the smallest unit of erase. Data can only be written to an empty (erased) page, and erasing is done at block granularity – this means flash **cannot directly overwrite** existing data in-place. Moreover, NAND flash has a limited endurance (each block can only be erased a certain number of times before it wears out) [8] [9]. These constraints require the SSD to perform additional work: if new data comes in for a page that is already written, the drive will write the data to a new physical page and later erase the old location during garbage collection. Flash memory operations are fast but still much slower than CPU or DRAM, so SSDs employ **parallelism** to boost throughput: multiple flash chips can operate concurrently on different channels. Most SSDs have **multiple flash dies** spread across channels so that the controller can issue many operations in parallel [5]. For example, an SSD might have 8 channels, each with 2 or 4 chips; the controller can read or write from several chips at once, dramatically increasing aggregate I/O rates. As research notes, *"most SSDs are composed of multiple NAND flash memory chips, and these chips operate in parallel by being tied to a channel and a way interface"* [7]. This internal parallelism is a key reason SSDs achieve high IOPS (input/output operations per second) – the device can handle many flash operations concurrently across the channels and chips.

Controller and Cache: The SSD controller typically includes one or more embedded processors (often ARM cores or similar) to run firmware, plus dedicated hardware logic for performance-critical tasks (like ECC calculation or encryption). There is usually a **DRAM or SRAM buffer cache** inside the SSD used to stage data and metadata [10] [11]. For writes, the cache can buffer incoming data so that the controller can batch and coalesce writes to flash (since programming a flash page is relatively slow) [10]. Read data may also be cached. This cache improves performance but introduces a risk: if power is lost, data in volatile cache could be lost before being written to flash. Therefore, enterprise SSDs often include **power-loss protection (PLP)** capacitors or other techniques to flush or protect data in cache on power failure [11]. The size and presence of a cache varies – high-performance SSDs have separate DRAM chips (e.g. DDR3/DDR4) for caching, while some lower-end SSDs may integrate a smaller cache on-chip or even rely on host memory (the latter is seen in technologies like HMB, Host Memory Buffer, for NVMe).

Flash Translation Layer (FTL) and Firmware Functions

One of the most important pieces of SSD controller firmware is the **Flash Translation Layer (FTL)**. The FTL presents the abstraction of a linear block device with logical block addresses (LBAs) to the host, but under the hood it translates each LBA read/write to a physical location on the flash memory [12] [13]. This indirection is necessary because of how flash works – you can't update in place, so the FTL maintains a mapping table to keep track of where the latest copy of each logical block resides in the physical flash. Typically, this mapping table is stored in the SSD's DRAM for quick access (and periodically checkpointed to

flash for persistence) [14]. When the SSD powers on, the map is reconstructed from metadata stored in flash.

Wear Leveling: Because flash blocks have limited endurance (each can only sustain a certain number of program/erase cycles), the FTL's job is also to distribute writes evenly across the media – a process called **wear leveling**. If certain physical blocks were written repeatedly while others were unused, those "hot" blocks would wear out prematurely. The controller instead tries to **spread out writes** so that all blocks approach end-of-life together [8] [9]. It does this by moving data around: for example, if one area of the logical address space is written very frequently, the actual physical blocks backing those LBAs will be rotated periodically. There are two types of wear leveling: *dynamic wear leveling* which focuses on distributing new writes evenly, and *static wear leveling* which additionally moves even long-lived data occasionally so that every block gets exercise. *"One of the main goals of an SSD controller is to implement wear leveling, which distributes P/E cycles as evenly as possible among the blocks"* [9]. This sometimes means extra background copying of data (which contributes to write amplification), so the firmware balances wear leveling against other performance factors [15].

Garbage Collection: Due to out-of-place writes, the SSD will eventually accumulate invalid data (stale versions of overwritten files) occupying flash pages. The controller performs **garbage collection (GC)** to reclaim this space. In garbage collection, the firmware identifies a flash block that has a lot of invalid pages (data no longer needed), reads any still-valid pages from that block and writes them to a new block, updates the mapping, and then erases the old block to free it [3] [4]. This process runs in the background (often when the drive is idle or under light load) to ensure a pool of free blocks is always available for incoming writes. Efficient garbage collection algorithms aim to minimize the amount of data that must be moved – for instance, choosing blocks with the most invalid data to erase (to reduce copy operations), and coalescing fragmented updates effectively. GC is closely tied to write amplification: every time the SSD has to rewrite data internally, it adds extra writes beyond what the host sent. A well-designed FTL tries to keep write amplification low by doing smart garbage collection and by leveraging extra over-provisioned space.

Over-Provisioning and TRIM: Most SSDs ship with more flash memory than they expose to the user (for example, a "256 GB" SSD might actually have 275 GB of raw flash). This extra space, called **over-provisioning**, gives the controller breathing room for wear leveling and garbage collection – it provides a reservoir of free blocks and a buffer for replacing bad blocks as they fail. Greater over-provisioning generally improves performance and endurance at the cost of usable capacity [16]. Another important feature is **TRIM** (in ATA) or **UNMAP** (in SCSI) and the analogous **NVMe deallocate** command. These allow the operating system to inform the SSD when certain LBAs are no longer in use (for instance, when files are deleted or a partition is formatted). Receiving a TRIM for a range of LBAs lets the SSD mark those logical blocks as invalid in its mapping – meaning it can immediately treat the underlying physical pages as stale. This helps performance: with TRIM, the controller knows it doesn't need to preserve that data during garbage collection, freeing blocks more efficiently [17]. Frequent TRIMs allow the FTL to maintain a larger pool of free blocks, reducing write amplification and wear [18]. In summary, the FTL and firmware in an SSD implement a host of strategies – wear leveling, garbage collection, bad block management, error correction, and more – to make a flash-based drive function as a reliable block device.

Interface Protocols: SATA/AHCI, NVMe, and SCSI

From the host system's perspective, an SSD (or any block device) is accessed via a specific interface protocol. The common interfaces today are SATA (which uses the AHCI protocol for command submission), NVMe

(over PCI Express), and SCSI (used in SAS drives or via SATA-to-SCSI translation). These define how the operating system sends commands like read and write to the device and how data is transferred. Modern SSDs can be found behind all these interfaces: you can have SATA SSDs (speaking the AHCI/ATA command set), or NVMe SSDs, etc. The device's hardware internals are similar, but the interface affects performance and software integration.

SATA and AHCI (Advanced Host Controller Interface)

SATA is the successor to the older PATA/IDE interface and was traditionally used for hard drives and SSDs in PCs. The SATA interface (at 6 Gbps for SATA III) still uses the ATA command set (also known as the "ATA/SATA protocol"), but modern systems implement SATA controllers using the **AHCI** specification. AHCI is a standardized register interface for SATA host controllers, allowing the same driver to work with different vendors' SATA controllers [19]. AHCI introduced features like **Native Command Queuing (NCQ)**, which allows a SATA drive to accept up to 32 outstanding commands and internally reorder them for efficiency. For spinning disks, NCQ helped by scheduling reads/writes to minimize seek time. For SSDs, NCQ provides a modest level of parallelism – 32 concurrent commands – which the SSD's controller can then distribute across flash chips. However, AHCI has a single submission queue and was originally designed with the constraints of hard drives in mind [20] [21]. Even with NCQ, the single-queue, 32-command limit becomes a bottleneck for high-performance SSDs that could handle much more concurrency [22].

AHCI communicates with the system via the PCI bus (the AHCI host controller is a PCI device, often integrated into the chipset). When the OS issues a SATA command, it writes to registers and a command list in memory as defined by AHCI, and the controller forwards the commands over the SATA link to the device. AHCI thus acts as a kind of adapter translating memory-mapped I/O operations into SATA wire protocol packets. This extra layer adds some latency and overhead. It wasn't a problem when drives (HDDs) themselves had ~10 millisecond latencies, but with SSDs delivering data in microseconds, the AHCI overhead and limited queue depth started to stifle performance [23]. As one source notes, as SSD speeds climbed, "the AHCI standard becomes the bottleneck. Therefore, NVMe is made to break the bottleneck of SSD performance." [24]

Software-visible Behavior: SATA devices under Linux historically appear as `/dev/sdX` nodes and are managed by the **SCSI subsystem**. This is an interesting quirk: Linux treats SATA drives much like SCSI drives by using the **libata** layer, which implements a SCSI-to-ATA Translation (SAT) so that SATA commands are issued through the well-established SCSI stack [25] [26]. This design was chosen because the SCSI storage stack in Linux was mature and supported features (like tagged command queuing, multiple outstanding I/O, etc.) similar to SATA's capabilities [27]. In practice, when Linux loads the `ahci` driver (for a SATA AHCI controller), the `libata` library code presents the attached drive to the system as a SCSI disk device. Thus, even a SATA SSD will show up as `/dev/sda` and be handled by the SCSI disk (`sd`) driver and SCSI mid-layer in the kernel. The SCSI subsystem sends ATA commands to the drive via the SAT protocol. This is why you might see SCSI terminology in logs or utilities even when using SATA drives. (Windows took a similar approach at one point – many Windows SATA drivers presented drives as "SCSI" to the OS until native AHCI support became common [28].) The AHCI/AHCI controller itself is memory-mapped and typically managed by the OS with its own driver, but once set up, I/O commands flow through the standard storage stack.

In summary, **AHCI+SATA** provides broad compatibility (any OS that supports AHCI can use any SATA drive) but the single-queue design (32 commands deep) and legacy overhead (register fiddling and interrupts per command) limit performance scaling. It works well for desktop workloads but cannot fully utilize the

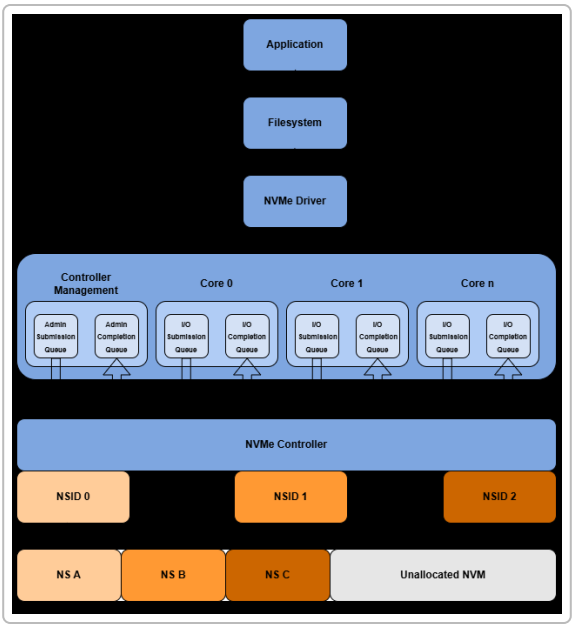
potential of modern flash. Measurements have shown that an AHCI SATA interface can sustain on the order of 100k IOPS and has higher per-I/O latency (e.g. ~6 microseconds just in command overhead) compared to newer interfaces [29]. This led to the development of NVMe for direct PCIe attached SSDs.

NVMe (Non-Volatile Memory Express)

NVMe is a storage interface and protocol designed from the ground up for fast, parallel flash storage. NVMe drives connect via PCI Express (either as add-in cards, M.2 modules, U.2 drives, etc.) and communicate using the NVMe protocol which is optimized for low latency and high concurrency. The NVMe specification allows a device to support **up to 64K submission queues, each up to 64K commands deep** [22] [23] – an enormous increase in parallelism compared to AHCI's single 32-deep queue. In practice, a typical NVMe SSD might use far fewer than that maximum (e.g. a few dozen or hundred queues), but the point is that NVMe's designers removed the bottleneck on command issuance. NVMe is also streamlined in terms of command processing: it uses a pair of simple ring buffers (submission and completion queue) in host memory, and the host signals new commands by writing a doorbell register (one per queue) to notify the device. There are only **a couple of MMIO register accesses needed per IO** (e.g. writing the doorbell and later reading a completion, often via an interrupt) [29] [30]. By contrast, AHCI had to write several registers for each command and had more overhead for each queued operation (one source notes AHCI needs 6 to 9 register accesses per command vs NVMe's 2) [31]. NVMe devices also support **MSI-X interrupts** and interrupt steering – each IO completion queue can use a different interrupt vector, allowing the interrupt to be delivered to the specific CPU core that submitted the I/O [32] [33]. This improves efficiency in multi-core systems by keeping I/O handling localized (reducing context switching and lock contention).

Protocol Efficiency: NVMe's command set is small and efficient – only a handful of commands (read, write, flush, etc. plus admin commands) rather than carrying decades of legacy features [34] [35]. The commands themselves are built for flash (for instance, NVMe has a command for "write zeros" or for TRIM/Deallocate, and it avoids legacy head/tail movement concepts). Because NVMe runs over PCIe, it also enjoys the huge bandwidth advantage of PCIe lanes – e.g. a 4-lane PCIe 4.0 SSD can use ~8 GB/s of bandwidth, far beyond SATA's 600 MB/s limit [36] [37]. The elimination of the SATA controller in the path (NVMe devices talk directly to the PCIe interface) cuts out intermediary latency. In fact, NVMe was shown to reduce latency by tens of microseconds compared to SAS/SATA. For example, one comparison noted SAS (a SCSI interface) might add ~25 µs protocol latency, whereas NVMe's overhead is on the order of 5 µs [38]. Even within the host, NVMe drivers use significantly fewer CPU cycles – one Seagate technical brief showed that to reach 1 million IOPS, an AHCI/SATA stack consumed ~27K CPU cycles whereas NVMe could do it in ~10K cycles [39] [40]. NVMe "gets out of the way" of the device's performance: it was architected so that the software interface would not become the limiting factor for fast flash media [41] [42].

Parallelism with NVMe Queues: NVMe's ability to have multiple queues means an OS can allocate **one or more submission/completion queue pairs per thread or per core** for I/O. In a multi-core system, this is a game-changer. The device can pull commands from multiple queues in parallel without any locking needed between CPUs on the host. Each completion queue can generate interrupts targeted to specific cores [32]. *Figure 2* illustrates how NVMe enables scalable multi-queue I/O processing



. In this diagram, the NVMe driver has set up per-core I/O queues (Core 0, Core 1, … Core n each have their own submission and completion queue), and the NVMe controller hardware services all those queues. There's also a separate Admin queue used for management commands (identify, etc.). The result is that multiple CPU cores can issue I/O to an NVMe SSD **simultaneously** without contending on a single software lock or single hardware queue [43] [44] . The device internally arbitrates between queues (e.g. simple round-robin or weighted schemes as defined in the NVMe spec) [45] [46] . By contrast, with AHCI, if multiple threads on multiple cores want to do I/O, their requests all funnel into the single queue – the driver must use a lock to coordinate command submission, which becomes a scalability bottleneck on multi-core systems [31] (AHCI effectively forces serialized command submission). NVMe removes that bottleneck. This massive parallelism is one reason NVMe drives achieve very high IOPS and can utilize many CPU cores efficiently for I/O. In fact, Linux's NVMe driver typically creates one hardware queue per CPU core to maximize parallelism [47] [48] .

Because NVMe was designed well after ATA/SCSI, it also incorporated features for modern needs, like end-to-end data protection, robust error reporting, and the ability to namespace the device (an NVMe controller can be split into multiple logical volumes or "namespaces" which are somewhat analogous to SCSI LUNs) [49] [50] . But at its heart, the key point is NVMe is a lightweight, high-performance protocol tailored for flash and PCIe.

SCSI (and SAS)

SCSI (Small Computer System Interface) is a long-standing storage protocol that has been used for everything from hard drives to tape drives. Today, the SCSI command set is primarily used in enterprise contexts, often over **SAS (Serial Attached SCSI)** or Fibre Channel, and also in protocols like iSCSI. SCSI is a feature-rich protocol with many commands and options (some consider it "heavyweight" relative to NVMe). A SAS SSD uses the SCSI command set over a SAS physical interface. SAS offers improvements over SATA in some areas: SAS supports dual-port drives, longer cables, and typically higher device counts in enterprise backplanes. SAS devices also allow a deeper command queue – up to **254** or **256 commands** outstanding to a single device [51] [52] . This is better than SATA's 32, but still a single queue. Additionally, a high-end SAS

HBA (host bus adapter) might manage multiple queues or have its own internal tag scheduling, but from the OS perspective, commands are sent one at a time up to the queue depth. Protocol-wise, SCSI has more overhead than NVMe: commands are delivered via a complex software stack (in Linux, the SCSI mid-layer issues SCSI CDBs to an HBA driver, which then communicates with the device). Each I/O goes through the SCSI command processing, which historically involved locking and less efficient interrupt handling. Thus, even though SAS SSDs can be quite fast, an NVMe PCIe SSD generally outperforms a SAS SSD of similar flash hardware, due to the leaner protocol. As an example, one vendor notes *"NVMe's architecture is highly parallel: it supports 65,000 I/O queues with up to 64K commands each, versus the single queue of SAS/SATA"* [53]. This parallelism and the removal of SCSI translation overhead give NVMe an edge in latency and throughput.

Software integration: In Linux, true SCSI or SAS devices (e.g. a RAID controller or SAS HBA with drives) appear as `/dev/sdX` as well, going through the same SCSI subsystem. So from the OS perspective, a SAS SSD and a SATA SSD might not look very different except that the SAS one might be handled by a different low-level driver. The SCSI stack in Linux has, in recent years, been updated to support a multi-queue mode as well (to better handle high IOPS devices). Starting with kernel 3.17, the SCSI core can use the blk-mq (multi-queue block layer) infrastructure [54]. This means even SCSI/SAS devices can benefit from reduced locking in the kernel I/O path, although they still ultimately feed into the device's single queue. SCSI's richness (e.g. extensive command set for reservations, scatter-gather lists, etc.) made it a workhorse of enterprise storage for decades, but it wasn't designed for microsecond latency. NVMe, arriving later, intentionally left out a lot of legacy baggage to streamline command processing [34] [35].

Comparing AHCI, NVMe, and SCSI – Parallelism and Efficiency

The differences between legacy interfaces like AHCI (SATA) and modern NVMe are stark, and have direct implications for performance. **Table 1** summarizes some key comparisons between AHCI and NVMe

Table 1. SSD protocol comparison between AHCI and NVMe

SSD protocol comparison	AHCI	NVMe
Maximum Queue Depth	1 queue 32 commands per queue	65535 queues 655336 commands per queue
Uncatchable Register Accesses	6 per non-queued command 9 per queued command	2 per command
MSI-X and Interrupt Steering	Single interrupt No steering	2048 MSI-X interrupts
Parallelism and Multiple Threads	Synchronous lock is required to issue a command	No lock
Efficiency for 4KB Commands	2 serialized host DRAM fetched for command parameters	Get command parameters in one 64-byte fetch

. NVMe supports orders of magnitude more commands in flight (e.g. 64K queues * 64K entries) than AHCI (1 queue * 32 entries) [22]. NVMe also requires far fewer CPU register interactions per IO (modern CPUs can even fetch an NVMe command entry in one 64-byte PCIe transaction) whereas AHCI's mechanism requires

multiple programmed IO steps [31]. NVMe devices can use up to thousands of interrupt vectors (allowing interrupt affinity to cores), compared to AHCI's single interrupt line [31]. The ability for NVMe to avoid locks in the I/O submission path (each core can ring its own doorbell without contending) is a big advantage on multi-CPU systems [31]. In practical terms, NVMe's design yields higher IOPS and throughput. For example, NVMe drives today can exceed **1–2 million IOPS**, whereas SATA/AHCI tops out around **100k IOPS** in many cases [55] [56]. Sequential throughput is also higher simply by virtue of PCIe bandwidth (NVMe drives routinely offer 3–7 GB/s, saturating PCIe x4 links, compared to SATA's ~0.6 GB/s limit) [36].

When it comes to **protocol latency**, NVMe has a clear edge. By cutting out intermediate layers, NVMe achieves lower latency on each IO. A SATA SSD might have ~100 µs latency for a random 4K read (which includes protocol/software overhead), while an NVMe SSD might do it in ~20–30 µs under ideal conditions [57]. The flash media itself also contributes latency (tens of microseconds to read from NAND), but NVMe minimizes everything on top of that. SCSI (SAS) typically falls in between – faster than SATA, but slower than NVMe. One source from Pure Storage noted SATA SSDs often see 100–200 µs, SAS a bit lower, and NVMe in the tens of µs range for latency [57]. Another source points out NVMe removed about *15–25 µs* of software overhead compared to SAS [38].

System Integration: NVMe's design did require operating systems to adopt new drivers and block layer enhancements (which happened around 2014–2015 in Linux, and similar time frame in Windows). In contrast, a big appeal of AHCI was that it could reuse existing OS models (SATA drives could use the same driver interface as older IDE drives, etc.). But the industry quickly moved to support NVMe because the performance benefits were compelling. NVMe is now standard in most OSes, and it coexists – you can have both SATA and NVMe SSDs in the same system. SCSI remains relevant mainly for enterprise and compatibility; NVMe is even making inroads in those domains (e.g. NVMe-oF for networks, NVMe drives in servers instead of SAS drives). In fact, NVMe was envisioned as a replacement for both SATA/AHCI and for SCSI in many cases, simplifying the software stack for high-performance storage [58] [59]. Linux did at one point experiment with a translation layer to present NVMe drives through the SCSI subsystem (to reuse things like device-mapper multipathing), but this was eventually removed due to the impedance mismatch and performance costs [60]. In summary, AHCI vs NVMe is a trade-off of legacy compatibility versus performance: AHCI/SATA provided a bridge from HDDs to early SSDs, but NVMe was needed to fully unleash flash potential. SCSI, being older, shares some of AHCI's issues (limited queueing, higher overhead), and thus NVMe is considered the future for direct-attached storage, especially in high-performance scenarios.

Linux Block Layer Integration and I/O Scheduling

In the Linux kernel, block devices are handled by the block I/O layer, which includes request queues and scheduling mechanisms. The introduction of fast NVMe devices prompted a re-design of this block layer to better handle parallelism. Historically, Linux used a single request queue per block device, and I/O schedulers (like CFQ, Deadline, or Noop) would merge and reorder requests primarily to optimize HDD access patterns. However, a single queue became a bottleneck when a device can handle hundreds of thousands of IOPS across many cores [61] [62]. Linux addressed this with the **Multi-Queue Block I/O (blk-mq)** subsystem introduced around kernel 3.13 and improved in subsequent releases [63] [64]. With blk-mq, the block layer can maintain **multiple software queues** and map them to **multiple hardware dispatch queues**, aligning with devices that support multiple hardware queues (like NVMe) [65] [66]. Each CPU (or each core) can have its own software queue for a device, which feeds into one of the device's hardware queues – thereby avoiding contention on a single queue lock. The NVMe driver was designed to take full advantage of this: it informs the block layer how many queues the device supports (often as many as the

number of CPUs), and the blk-mq framework sets up that many queues [47] [48]. In fact, the Linux NVMe driver typically creates one queue pair per core (1:1 mapping of software queue to hardware queue) to maximize parallel throughput [47] [48]. This means that if you have, say, an 8-core system, the NVMe block device `/dev/nvme0n1` will be serviced by 8 submission/completion queues in the kernel, each mostly handled by its respective core. The completion interrupts from each queue can be routed to the right core thanks to NVMe's support for multiple MSI-X interrupts [32] [67]. The result is a drastic reduction in locking and CPU overhead per I/O – Linux can scale to millions of IOPS across cores as long as the device can handle it.

By contrast, a SATA SSD on AHCI (which has one hardware queue) will end up with effectively one hardware dispatch queue. Even though Linux's SCSI layer was adapted to blk-mq (often called *scsi-mq*), the physical limitation of one queue of depth 32 means it cannot issue more than 32 operations at a time to the device. The blk-mq framework can still help distribute submission work across cores to some extent (e.g. 32 requests can be split among cores), but ultimately they funnel into the single hardware queue. There's also still a need to synchronize access to that single queue's doorbell register (i.e. a lock in the AHCI driver when submitting commands). Thus, SATA devices do not scale with cores in the same way – they tend to get capped in IOPS by that single queue and the overhead of context switching if many threads compete for it.

I/O Scheduling: The role of the I/O scheduler has changed with SSDs. For rotating disks, schedulers like CFQ (Completely Fair Queuing) or Deadline rearranged and merged requests to minimize seek time and ensure fairness. SSDs have constant seek cost (effectively zero seek time), so reordering for performance isn't needed to reduce seek overhead. In fact, excessive reordering could hurt performance by preventing the SSD from working on requests in parallel (though that's less of an issue if depth is maintained). With fast SSDs, especially NVMe, the emphasis shifted to reducing software overhead. Linux introduced the **"noop"** scheduler (which does no reordering, just FIFO) and **"mq-deadline"** (a lightweight scheduler for multi-queue) as alternatives. In modern Linux setups, **multi-queue devices default to the** `none` **scheduler**, which effectively means the kernel does minimal scheduling and lets the device handle queueing [68]. For NVMe, the typical default is "none" or "mq-deadline" depending on the distro, but in many cases using none (no scheduler) yields the best throughput by removing any extra queuing latency in the OS. The NVMe controller itself can handle command prioritization and fairness via its arbitration mechanism if needed [46] [69]. SATA SSDs, since they go through the SCSI layer, might by default use the "mq-deadline" scheduler in newer kernels (or "cfq" in older ones), but many users switch them to "noop" as well, since there's little benefit to complex scheduling for an SSD – the priority is to deliver requests to the device as quickly as possible. In fact, Samsung and other vendors often recommend using simpler I/O schedulers for SSDs. Additionally, merging of requests (coalescing adjacent writes) is less critical given high random performance, although merging can still reduce protocol overhead if the OS can combine contiguous requests into one larger request. The blk-mq design still supports request merging, but it's done per software queue.

I/O Path Differences: With NVMe devices on Linux, an I/O request from an application might go through the filesystem to the block layer, get placed into a software queue on the same CPU core, then almost immediately dispatched to the NVMe hardware queue for that core. The NVMe driver writes to the device's doorbell register to notify of new commands. The device DMA-transfers the data and then posts a completion entry in the queue and triggers an interrupt to that CPU core. The latency added by the OS is very low – on the order of a few microseconds or less. With a SATA SSD, the same I/O goes through the filesystem and into the single request queue of the SATA device (which might be processed by a block-layer scheduler). The libata layer then packs it into an AHCI command FIS structure in memory. The AHCI

controller (which might be part of the PCIe southbridge) is notified via a register write. It fetches the command and sends it over the SATA cable to the drive. The overhead in the OS is slightly more, and the interrupt handling is less granular (there's typically one interrupt for the host adapter that all completions use). So while both SATA and NVMe SSDs benefit from the lack of seek delays, the NVMe's end-to-end path is shorter and more parallel.

Impact on Performance and Decisions: The internal behavior of SSDs – particularly their need for parallelism and how they handle queued commands – strongly influences how we design systems. For instance, to get the best performance from an NVMe SSD, you might use multiple threads or asynchronous I/O so that many requests are in flight concurrently (leveraging those multiple queues and the device's ability to work on many things at once). The Linux I/O stack with asynchronous I/O (io_uring or libaio) can submit many requests without waiting for each to finish, which pairs well with NVMe. By contrast, with a SATA SSD, after 32 outstanding requests, additional I/Os will start to queue up in the OS, so beyond a point you don't gain much by adding more parallel threads – you hit the device queue limit. Also, because NVMe devices are so fast, other parts of the system become bottlenecks: CPU overhead, PCIe bandwidth, or application-level processing. It's not uncommon that an NVMe drive's latency is low enough that doing a system call per I/O becomes a limiting factor (hence Linux has been working on things like io_uring to submit many I/Os with minimal syscalls). In storage scheduling for multi-tenant environments, NVMe's multiple queues also open the door to isolating workloads – e.g. using different NVMe queues or namespaces for different applications to prevent interference, something that was harder to achieve with single-queue devices.

Finally, features like **NVMe Streams or Zoned Namespaces** (advanced concepts beyond scope here) are being introduced to give the host more control over data placement, which can reduce write amplification by aligning with the SSD's internal patterns. These rely on the close integration between host software and the device's capabilities, again highlighting how modern SSDs blur the line between device and host responsibilities for performance.

Conclusion

Modern block storage devices have evolved far beyond the simple spinning disk – they are complex systems combining hardware and software to manage the quirks of flash memory and to deliver extreme performance. SSD controllers handle flash translation, wear leveling, error correction, and caching to present a reliable, high-speed block device to the host. Interface protocols like NVMe capitalize on the parallel nature of flash and the high throughput of PCI Express, enabling millions of IOPS and low latency by removing the bottlenecks of older standards like AHCI and SCSI. In Linux, the storage stack has adapted with a multi-queue block layer that maps naturally onto devices with dozens of hardware queues, ensuring that the operating system can keep up with the devices. When integrating SSDs and especially NVMe drives into a system, one must consider these factors – for optimal performance, use the appropriate interface (NVMe wherever possible for high performance), use modern I/O APIs and schedulers that minimize software overhead, and be mindful of the drive's background processes (for example, provisioning enough spare area and enabling TRIM to help the device's garbage collection). In summary, understanding how an SSD works internally and how it interfaces with the OS can help in making informed decisions to fully harness the potential of today's storage technology, achieving a balance of speed, durability, and efficiency.

Sources: The information above was synthesized from technical sources including the NVM Express organization's documentation on SSD architecture [70] [6], comparisons of NVMe and AHCI by industry

experts [22] [23], Linux kernel documentation and wiki on the block layer [65] [71], as well as educational articles on SSD internals (flash memory, FTL) [9] [12]. These references and others are cited inline to provide detailed backing for the concepts discussed.

---

[1] [2] Solid State Drive Primer # 9 - Controller Architecture - Controller Block Diagram
https://www.cactus-tech.com/resources/blog/details/solid-state-drive-primer-9-controller-architecture-controller-block-diagram/

[3] [4] [5] [6] [10] [11] [16] [70] NVMe™ Form Factors Blog Series Part II: “NVMe Building Blocks – Controller, Buffer Memory, Media and Form Factors” - NVM Express
https://nvmexpress.org/nvme-form-factors-blog-series-part-ii-nvme-building-blocks-controller-buffer-memory-media-and-form-factors/

[7] Layout of NAND flash memory-based SSD. | Download Scientific Diagram
https://www.researchgate.net/figure/Layout-of-NAND-flash-memory-based-SSD_fig1_346417707

[8] [9] [12] [13] [14] [15] Coding for SSDs – Part 3: Pages, Blocks, and the Flash Translation Layer | Code Capsule
https://codecapsule.com/2014/02/12/coding-for-ssds-part-3-pages-blocks-and-the-flash-translation-layer/

[17] [18] Does it matter when I activate trim on SSD? - Super User
https://superuser.com/questions/1615010/does-it-matter-when-i-activate-trim-on-ssd

[19] [24] AHCI vs. NVMe - Phison Blog
https://phisonblog.com/ahci-vs-nvme-the-future-of-ssds-2/

[20] [21] [49] [50] Overview of NVMe Architecture | linux
https://blogs.oracle.com/linux/overview-of-nvme-architecture

[22] [23] [29] [30] [36] [37] [55] [56] Understanding SSD Technology: NVMe, SATA, M.2 - Kingston Technology
https://www.kingston.com/en/ssd/what-is-nvme-ssd-technology

[25] [26] [27] [28] [58] [59] [60] Why does the Linux SCSI subsystem drive hardware not obviously related to SCSI? - Super User
https://superuser.com/questions/1824698/why-does-the-linux-scsi-subsystem-drive-hardware-not-obviously-related-to-scsi

[31] phisonblog.com
https://phisonblog.com/wp-content/uploads/2021/01/1013920_PhisonSSDControllerWP_032421.jpg

[32] [33] [34] [35] [41] [42] [43] [44] [67] sata-io.org
https://sata-io.org/sites/default/files/documents/NVMe_and_AHCI_long.pdf

[38] SAS vs. NVMe: The future of the two key storage interfaces - Tekmart
https://tekmart.co.za/t-blog/sas-vs-nvme-the-future-of-the-two-key-storage-interfaces/?srsltid=AfmBOooH8VDi0FVgUylPzWieD1FjCLwzayWTfc-0gJdjGAKizKFiCb41

[39] [40] Tech Brief: NVMe—Performance for the SSD Age >
https://www.seagate.com/content/dam/seagate/migrated-assets/www-content/product-content/ssd-fam/nvme-ssd/nytro-xf1440-ssd/_shared/docs/nvme-performance-tp692-1-1610us.pdf

[45] [46] [47] [48] [69] Title for USENIX Conference Paper: Sample First Page
https://www.usenix.org/system/files/conference/hotstorage17/hotstorage17-paper-joshi.pdf

[51] SAS vs SATA: Which Storage Interface Is Right for You? | HP® Tech ...
https://www.hp.com/us-en/shop/tech-takes/sas-vs-sata

52 What is queue depth and how does it work? - TechTarget
https://www.techtarget.com/searchstorage/definition/queue-depth

53 Comparing SAS, SATA, NVMe, and CXL SSDs in 2025 - Server Simply
https://www.serversimply.com/blog/comparing-sas-sata-nvme-and-cxl?srsltid=AfmBOopkyPiOny47v70o9UWjKQ2BpKsEAcfNlT6Hk092ZJcHLCSoW09t

54 63 64 65 66 71 Linux Multi-Queue Block IO Queueing Mechanism (blk-mq) Details - Thomas-Krenn-Wiki-en
https://www.thomas-krenn.com/en/wiki/Linux_Multi-Queue_Block_IO_Queueing_Mechanism_(blk-mq)_Details

57 What Is NVMe? | Pure Storage
https://www.purestorage.com/au/knowledge/what-is-nvme.html

61 62 kernel.dk
http://kernel.dk/blk-mq.pdf

68 Chapter 12. Setting the disk scheduler | Red Hat Enterprise Linux | 8
https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/8/html/monitoring_and_managing_system_status_and_performance/setting-the-disk-scheduler_monitoring-and-managing-system-status-and-performance