Exploiting Directly-Attached NVMe Arrays in DBMS

Gabriel Haas, Michael Haubenschild†, Viktor Leis
Friedrich-Schiller-Universität Jena
{gabriel.haas,viktor.leis}@uni-jena.de
Tableau Software†
mhaubenschild@tableau.com†

ABSTRACT

PCIe-attached solid-state drives offer high throughput and large capacity at low cost. Modern servers can easily host 4 or 8 such SSDs, resulting in an aggregated bandwidth that hitherto was only achievable using DRAM. In this paper we study how to exploit such Directly-Attached NVMe Arrays (DANA) in database systems. We find that DANA presents new challenges that require rethinking the way I/O operations are performed at both the database and operating system layer.

1. INTRODUCTION

The performance of database systems depends crucially on how well these systems are optimized for the hardware they run on. For example, after decades of rapidly-dropping DRAM prices, minimizing the number of disk I/O operations became less important. The database community reacted by redesigning and rethinking the traditional architecture—resulting in novel designs like main-memory database systems. This shows the importance of keeping track of hardware trends, as these must be taken into account when designing and developing high-performance database systems.

Figure 1 shows the capacity per dollar of main memory (DRAM) and storage (disk and flash) since the year 2000. Two observations are apparent from the figure. First, the growth of DRAM capacity has slowed down considerably since about 2011. Second, flash (i.e., NAND-based solid-state drive) capacity has grown faster than DRAM and disk. At the end of 2019, flash is 5–10 times more expensive than disk, and 20–40 times cheaper than DRAM.

Flash has become attractive not only in terms of capacity but also in terms of performance. Until recently, solid-state-drives (SSD) were usually attached through the SATA interface, which has a bandwidth of less than 0.6 GB/s. Modern solid-state drives, in contrast, are directly attached to PCIe (through M.2 or U.2 and the NVMe interface) rather than SATA. PCIe unlocks the abundant internal parallelism of SSDs, which internally consist of dozens of flash chips. As a result, a single commodity SSD can access over 3 GB/s using 4 PCIe 3.0 lanes. Current server CPUs have between 48 or 64 PCIe lanes; Thus, even after leaving some lanes for networking, one can directly connect 8 SSDs—resulting in an aggregated bandwidth of about 25 GB/s per socket. Amazon AWS, for example, offers instances with 8 NVMe SSDs. We call such a configuration of multiple high-performance PCIe SSDs *Directly-Attached NVMe Array (DANA)*.

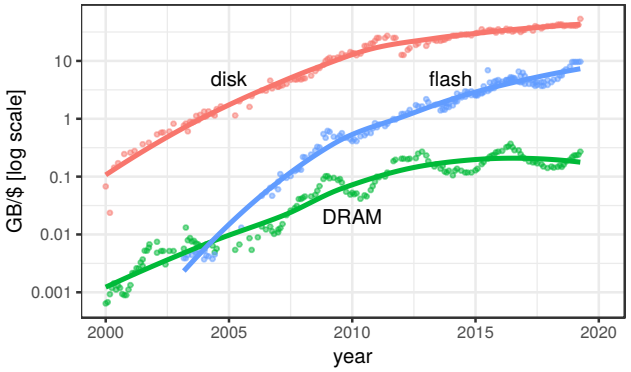


Figure 1: Historical disk, flash, and DRAM capacity per dollar.
data source: https://jcmit.net/memoryprice.htm

To put a DANA bandwidth of 25 GB/s into perspective, let us compare it with other technologies. Eight 1 TB high-speed SSDs cost around $2000, which would also be enough to buy twenty 4 TB disks, three 128 GB modules of byte-addressable persistent memory ("PMem"), or twelve 32 GB DRAM modules instead. All of these alternatives can be connected to a single CPU socket. As Table 3 shows, in terms of bandwidth, even 20 magnetic disks are still 12 times slower than an eighth-way DANA. Persistent memory ("Intel Optane DC Persistent Memory") achieves slightly higher read bandwidth, but is slower for sequential writes. Only DRAM is still clearly faster than DANA. However, DRAM is 20 times more expensive in terms of capacity, and the bandwidth gap between DRAM and flash bandwidth is about to disappear. The just-launched AMD Rome platform supports PCIe 4.0, which doubles the bandwidth per lane, and has 128 lanes per socket.

Table 1: What do $2000 buy?
based on commodity prices, PMem and DRAM performance from [17]

	configuration	capacity [TB]	bandwidth read [GB/s]	bandwidth write [GB/s]	random reads [M/s]
disk	20×4 TB	72.8	2	2	0.002
DANA	8×1 TB	7.3	25	20	4
PMem	3×128 GB	0.4	37	10	155
DRAM	12×32 GB	0.4	92	80	1,500

9th Biennial Conference on Innovative Data Systems Research (CIDR '19) January 13-16, 2019 , Asilomar, California, USA.

Because DANA is much faster than disk and much cheaper than DRAM and PMem, it is the ideal storage technology for large scan-oriented OLAP databases—as well as HTAP workloads for which the transactional working set fits into main memory. This has consequences for database system architecture. Data should be stored primarily on flash, with DRAM being used as a cache for frequently-accessed data items and transient query processing structures like hash tables. Disk is mainly useful for backups and for archival storage, and, at current prices, the role of PMem seems unclear for scan-oriented workloads.

One important remaining question about DANA-backed database systems is whether the promised performance is achievable in practice. After all, one major difference between DRAM/PMem versus flash/disk is that the latter require I/O system calls. Before an I/O request scheduled by the database system appears at the SSD, it goes through many OS layers, including buffering, the file system, RAID, and queuing. As we show in this paper, handling hundreds of simultaneously-scheduled I/O requests and millions of I/O requests per second is a major challenge.

The goal of this paper is to determine how to best exploit DANA in database systems. One practical difficulty is that operating systems offer many different ways to execute I/O operations. To find a good configuration, we perform an extensive experimental study using 4 high-performance SSDs on Linux. Using a challenging but realistic I/O workload, we quantify the impact of (1) OS buffering (page cache, `O_DIRECT`), (2) the I/O interface (pread/pwrite, libaio, io_uring), (3) the file system (ext4, Btrfs, XFS, F2FS), and (4) RAID (`md`, hardware RAID). In addition, we investigate SPDK, a user-space I/O stack that bypasses the operating system completely by directly accessing the PCIe interface.

The rest of the paper is organized as follows. After discussing related work in Section 2, we describe the hardware, workload, and software setup for our experiments in Section 3. The core of the paper is Section 4, which evaluates the OS I/O stack and SPDK. Based on these experiments, we discuss the implications of DANA for database system design in Section 5 and issue specific recommendations. We summarize the paper and mention future work in Section 6.

2. RELATED WORK

Optimizing database systems for flash is a topic that has been studied for more than a decade [9, 4, 2, 13, 12, 7]. However, as Figure 1 illustrates, in the past, flash was expensive and therefore generally thought of as an additional layer in the storage hierarchy. Since then, flash has replaced disk for most use cases, and modern PCIe-attached NVMe SSDs are about 6 times faster in terms of bandwidth than SATA-attached SSDs.

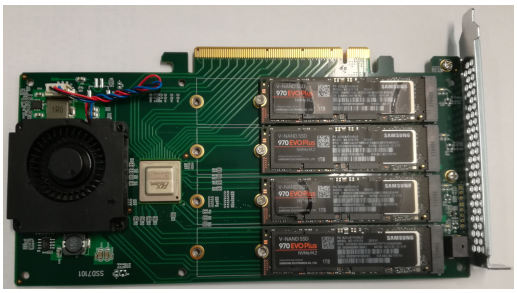
NAND flash has peculiar physical properties that are very different from magnetic disks. Recent work shows ways of exploiting these properties to improve performance, predictability, and durability [3, 14, 15, 6]. Nevertheless, given the proliferation of large data sets, we argue that more research on flash is required. In particular, we are not aware of any prior work that systematically evaluates *arrays* of high-performance flash devices or shows how to use them effectively. As we show in this paper, a DANA is nothing like a traditional storage device and requires new thinking about data access.

This paper is part of the LeanStore [10] project. LeanStore is a high-performance storage engine for modern hardware that achieves (almost) zero-overhead buffer management. This work is the first step to optimize LeanStore for DANA. Our findings are also highly-relevant to other recently-proposed storage engines [8, 16, 1].

3. EXPERIMENTAL SETUP

3.1 Hardware

Unless otherwise noted, we use 4 high-end, consumer-grade Samsung 970 EVO Plus 1TB SSDs, which are specified at 3,500 MB/s read and 3,300 MB/s write throughput[1]. To connect the 4 M.2 SSDs to our system, which has only 2 M.2 slots, we rely on the HighPoint SSD7101A-1 controller:



The controller hosts the four SSDs and is itself connected using 16 PCIe 3.0 lanes. It either simply exposes the SSDs to the operating system as separate devices or serves as a hardware RAID controller. We validated that in the former mode, which we use for most experiments, the controller does not add any overhead. Our system has an Intel Core i9-9900K CPU with 8 cores (16 hyperthreads) and 64 GB of RAM.

3.2 Workload

From a high-level point of view, database systems perform the following classes of I/O operations:

- *scan*: The performance of table scans is determined by throughput (rather than latency of any individual I/O operation). The database system can (and should) asynchronously prefetch a large number of pages.
- *point read*: For certain read operations, for example synchronous index lookups, latency is crucial and prefetching is not applicable.
- *WAL*: On transaction commit, database systems have to flush the tail of the write-ahead-log (WAL) to persistent storage. This is usually implemented using the `write` and `fdatasync` system calls. Commits are latency-critical.
- *background write*: Database systems buffer writes in main memory, and a background writer process continuously writes out dirty pages at a steady pace. Latency does not matter as long as the desired write rate is achieved.

Because we focus on I/O, we only access the data from SSD without processing it. However, we did measure CPU utilization to determine whether enough CPU cycles are available for actual query processing. For the scan, we schedule requests in batches of 32 and a total of 128 I/O requests simultaneously. We measure the latency of point reads by issuing a single read request at a time without waiting in-between. For the background writer, we specify a target write rate of 250 MB/s. The WAL writer also has a target write rate of 250 MB/s, but performs an `fdatasync` operation after every write. This configuration is summarized in Table 3.

In most experiments, we run the four workload I/O patterns simultaneously. Let us nevertheless mention the individual numbers to establish upper bounds. At an I/O depth of 128, we measured a

[1]Only the 6-42 GB SLC write cache achieves 3,300 MB/s. The steady write speed is 1,700 MB/s. See Section 4.8 for more details.

Table 2: The OS I/O stack. Entry #1 uses fdatasync for WAL, read/write system calls, OS buffering, and the ext4 file system. The following entries disable these features one by one, arriving at unbuffered block device access using asynchronous system calls.

	#	description	Section	I/O interface	scan MB/s	point MB/s	bg. wr. MB/s	WAL MB/s	total I/O MB/s	CPU cycles/byte
RAID 0 / file system / OS buffer / synchronous / fsync	1	baseline	4.1	pread/pwrite/sync	3,503	27	250	6	3,786	6.92
	2	− `fdatasync`	4.2	pread/pwrite	5,055	39	250	250	5,595	10.52
	3	+ async. I/O	4.3	io_uring	3,439	118	252	251	4,059	7.61
	4	+ `O_DIRECT`	4.3	libaio	4,464	74	250	249	5,036	1.01
	5	− file system	4.4	libaio	5,443	52	251	251	5,997	0.76

Table 3: Workload summary.

type	I/O operation	pattern	I/O depth	rate limit
scan	read	random	128	-
point read	read	random	1	-
bg. write	write	random	1	250 MB/s
WAL	write+sync	sequential	1	250 MB/s

maximum throughput of 6.8 GB/s for the scan. For the point reads, we achieved up to 10,560 page reads per second (165 MB/s), and for the synchronous WAL writes we measured 800 to 900 write operations (followed by an `fdatasync`) per second (12-14 MB/s). Running all four I/O classes simultaneously simulates an HTAP workload.

3.3 Software Setup

We use Linux 5.2.2 and simulate the four I/O classes described above by using the open-source I/O benchmarking tool `fio` in version 3.15. Unless otherwise noted, we use a page size of 16 KB, Linux' `md` software RAID 0 implementation, and the `ext4` file system, as it is the default under Linux.

To limit the impact from internal state, the SSDs are `blkdiscard`ed and freshly initialized after every run of the experiment. To reduce other noise and the influence of temperature, the SSDs are always given time to cool down between experiments.

4. A DEEP-DIVE INTO THE I/O STACK

The goal of this paper is to find out how to best use a DANA for database workloads. To do this, we run the workload described in the previous section using different OS I/O stack configurations—measuring I/O performance and CPU utilization. We start with a fully-featured setup, and gradually disable OS features to understand their impact on performance.

4.1 The Traditional I/O Stack

We start with a configuration that mimics the traditional way of how most disk-based database systems like PostgreSQL perform I/O by default. In addition to their own buffer pool, many systems also rely on OS buffering. Furthermore, today most systems store data on a file system rather than directly managing the block device. We thus use an OS cache of 10 GB and store both the 400 GB data file and the 100 GB WAL file on an `ext4` file system. Data is accessed using the `pread` and `pwrite` system calls from separate threads (we use 128/1/1/1 threads for scan/point-read/bg.-write/WAL). Although using 128 threads for the scan may seem excessive, with `pread` this is necessary to get good throughput on a modern SSD.

Entry #1 of Table 2 summarizes the results for the baseline configuration. Only the background writer achieves its desired rate of 250 MB/s. The scan (3.5 GB/s) and the point reads (27 MB/s, mean latency of 578 µs) fall below expectations. However, the biggest underachiever is the WAL thread. It only manages to achieve 6 MB/s (mean latency of 2.6 ms per WAL flush), which is much slower than the desired rate of 250 MB/s. Even though a poor result is expected in this setting, as typical write latency of NAND flash is already around 1 ms [3], it is still surprising that we can only achieve a fraction of that number.

4.2 Removing `fdatasync` From the Critical Path

The underwhelming results of the first experiment are caused by the frequent `fdatasync` system calls. `fdatasync` is a highly-invasive synchronization call (`fsync` even more so) that not only limits the performance of commits, but also slows down the other I/O types. One solution is to avoid frequent `fdatasync`s in the first place using a persistent memory buffer. Persistent memory or NVDIMMs (flash-backed DRAM) are ideal for flushing the tail of the log[2], because they have very low latency and are accessed directly using load/store CPU instructions rather than I/O system calls. This reduces the commit latency from milliseconds to about 0.3 µs [17]. Only a small amount of persistent memory is necessary, because after commit, the WAL data can be asynchronously written back to flash. In effect, this approach transforms synchronous log writes to asynchronous background writes—while improving commit latency significantly.

To simulate this approach, we removed the `fdatasync` calls from our workload without changing any other settings (this assumes that the overhead of flushing to persistent memory is negligible in comparison with the latency of NAND flash). As Table 2 shows, this small change has a significant positive effect, not only on the WAL thread, but on the entire workload: The desired WAL write rate of 250 MB/s is now achieved and point I/Os are faster by 44% despite scan performance is simultaneously increasing by the same factor.

4.3 CPU Utilization and Asynchronous I/O

At this point, we achieve a total I/O bandwidth of 5.5 GB/s but have very high CPU utilization. Measuring the system-wide CPU utilization using `perf`, we found that on average more than 10 cycles are necessary just to load one byte into the CPU. To access the 5.5 GB/s from our SSDs, our 8-core high-frequency CPU is fully utilized.

This is a major problem because, as we mentioned in the introduction, one would like to support 8 or 16 SSDs rather than just 4: At 10 CPU cycles per byte, a hypothetical system with eight 3 GB/s

[2]Another low-latency alternative is to ship the WAL to another machine using RDMA [18].

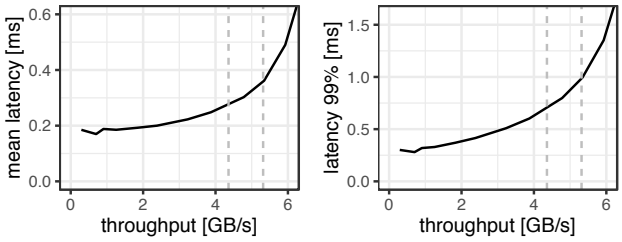


Figure 2: Read latency vs. throughput for random reads.

SSDs would require *86* cores at 3 GHz. And this still ignores the fact that there are also CPU cycles needed for query processing, not just for fetching the data. To make DANA-based database systems competitive with in-memory systems on scans, it is therefore crucial to reduce the CPU load by at least one order of magnitude.

To reduce CPU load, one promising approach is to use *asynchronous* rather than blocking I/O system calls. So far, we implemented the scan using 128 threads where each thread uses the blocking `pread` system call. With asynchronous I/O, a single thread can schedule all these I/O operations at once. Linux offers two ways for doing this: `libaio` is the traditional interface (since kernel 2.6), but does not support OS buffering. The very recent `io_uring` approach (since kernel 5.1), on the other hand, supports OS buffering, but is still under heavy development. We generally found the performance of both approaches very similar for our workload, and therefore show numbers for whichever method works in that particular configuration.

Entry #3 in Table 2 shows that enabling `io_uring` while still using OS buffering does not improve CPU utilization substantially. Scan performance actually decreases because a single scan thread with an I/O depth of 128 cannot keep up. Entry #4 shows that, in addition to using asynchronous I/O, one has to disable OS buffering (and use `O_DIRECT` mode) to reduce CPU load to 1.01 cycles per byte. At this point it becomes possible to utilize 8 SSDs, though it would still require more than eight 3 GHz cores

4.4 File System

After we disable OS buffering, the next source of overhead is the `ext4` file system. As entry #5 in Table 2 shows, the impact is significant. In comparison with ext4, direct block device access increases scan throughput by 22% and decreases the CPU cycles per byte by 33%.

The only part of the workload where disabling the file system seemingly hurts performance are point reads, which decrease from 74 to 54 MB/s. In fact, the root cause for the increased point read latency is not the file system: As the number of I/O requests increases, reads will have higher latencies and it becomes more likely that a read is stalled by other earlier I/O requests—causing longer queuing delays. To see this effect in isolation, we varied the I/O depth while measuring throughput and latency in a read-only experiment. As Figure 2 shows, mean latency increases by 80 µs, and 99th percentile latency increases even more by 300 µs. This trade-off between latency and throughput is a challenge for HTAP systems that concurrently run latency-critical point reads and large scans.

Although `ext4` is the most common file system on Linux, there are a number of alternatives. Table 4 shows the results for several other common file systems and compares them with direct block device access (without a file system). For our workload, XFS (entry #4b), default file system in Red Hat, clearly performs best. It has very little overhead in comparison to the configuration without

Table 4: File system impact. #5 is block device access.

#	file system	scan MB/s	point MB/s	write MB/s	total I/O MB/s	CPU cycl./byte
4	ext4	4,464	74	499	5,036	1.01
4b	XFS	5,395	53	502	5,950	0.79
4c	Btrfs	3,364	94	500	3,957	2.42
4d	F2FS	4,786	67	504	5,358	0.99
5	block	5,443	52	502	5,997	0.76

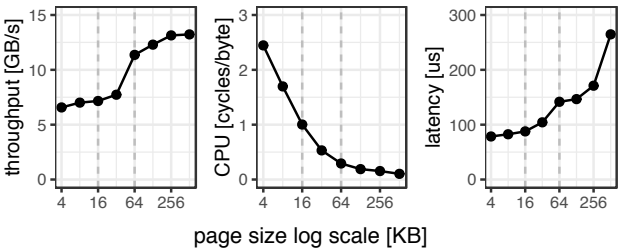


Figure 3: Impact of page size for random reads.

a file system (entry #5) both in terms of total I/O and CPU utilization. The other three file systems are significantly slower, with Btrfs performing worst, even when mounted with disabled copy-on-write. These results indicate that XFS, which has been designed for high-performance multi-threaded use cases, is a very good fit for DANA.

4.5 Page Size

So far, all experiments use a page size of 16 KB, which is the default page size of LeanStore [10]. The page size impacts not just point read latency, but also scan throughput because we use random rather than sequential I/O to simulate the scan—exploiting the fact that, in contrast to disks, SSDs are efficient at random access. This simplifies the design of SSD-optimized database system because it avoids the need to maintain the spatial locality of relations at the storage level. Disk-based database systems, in contrast, have to go to great lengths to cluster relations to get reasonable scan performance.

To evaluate whether this random-access design can achieve good throughput, we perform a read-only microbenchmark varying the page size. The results are shown on the left-hand side of Figure 3. Using a page size of 4 KB results in a read throughput of about 5 GB/s, which increases to 6 GB/s with the 16 KB setting used so far. A page size of 64 KB achieves a throughput of more than 11 GB/s, which is fairly close to the maximum throughput of 12 GB/s. These results indicate that for random scans, a page size of at least 64 KB should be used. Furthermore, as the center plot in Figure 3 shows, larger page sizes also significantly reduce CPU utilization because most in-kernel overhead is per system call, not per byte.

However, as the plot on the right-hand side of Figure 3 shows, larger page sizes also have a downside, namely higher latencies. A page size of 64 KB results in a 57% higher latency than with 16 KB pages. Thus, HTAP systems face the tradeoff between latency and throughput when deciding for a page size.

4.6 RAID

In all experiments so far, we used Linux' RAID 0 implementation. RAID 0 does not improve reliability, because it simply interleaves pages across devices, thereby exposing several SSDs as

Table 5: RAID impact. `md` is Linux software RAID, HW is hardware RAID. #7 is interleaved block device access.

#	RAID	scan MB/s	point MB/s	write MB/s	total I/O MB/s	CPU cycl./byte
5	`md` 0	5,443	52	502	5,997	0.76
5b	`md` 5	4,511	45	493	5,049	1.31
5c	`md` 10	4,974	50	498	5,522	0.89
6	HW 0	5,553	56	502	6,111	1.66
6b	HW 5	132	1	217	350	7.57
6c	HW 10	4,107	44	510	4,661	0.48
7	no RAID	5,665	53	499	6,217	0.76

one logical device. For a DANA of 4, 8, or even 16 devices, the probability of failure is non-negligible, and redundancy at the storage level becomes crucial. For 4 devices, RAID levels 10 or 5 are reasonable options, though both have downsides. RAID 10 has the obvious disadvantage of losing half the capacity and write speed, whereas RAID 5 has performance problems with random updates.

To determine the performance of different RAID levels and implementations on our workload, we compare Linux' software RAID implementation (`md`) with the hardware RAID options of our HighPoint SSD7101A-1 controller. The results for `md`, shown in Table 5, are as expected. RAID 0 (entry #5) has very little overhead in comparison with directly accessing the 4 block devices (entry #7). RAID 5 and 10, in contrast, result in a slowdown in terms of overall I/O throughput. These slowdowns are caused by additional I/O requests necessary to implement the RAID. RAID 10 doubles the number of physical writes, causing interference with scan performance. For RAID 5, the random background writes are the major issue, because a single random update results in an additional write of the parity page and two reads of the other pages in the RAID stripe. These additional I/O requests and the CPU cost of parity computation almost doubles CPU utilization on our workload in comparison with RAID 0.

Hardware RAID performs significantly worse than software RAID. The fact that the CPU utilization increases for RAID 0 (entry #6) in comparison with software RAID 0 (entry #5) may be surprising, but shows that our controller, which is marketed as a hardware RAID, in fact implements RAID using a proprietary kernel module in software. Let us note that, to the best of our knowledge, Highpoint is the only NVMe hardware RAID manufacturer.

To summarize, the performance results indicate that software RAID is preferable to hardware RAID. However, neither RAID level 10 nor level 5 are ideal for DANA. RAID 10 is not very economical as it wastes half the capacity. RAID 5, on the other hand, has high write and read amplification for random writes. Nevertheless, RAID 5 is a good option for those storage engines that avoid in-place updates.

4.7 User-Space I/O With SPDK

Given the performance and CPU utilization issues of the OS' I/O stack, it may look attractive to bypass the kernel and perform I/O directly from user space. This can be done using the *Storage Performance Development Kit (SPDK)*, a library that enables user-space access to NVMe SSDs and promises high performance and scalability. Instead of interrupts, SPDK relies on polling—one or more dedicated I/O threads manage the NVMe devices by constantly checking for events in a busy-waiting loop. Thus, SPDK replaces system calls with user-space polling, and intra-kernel locking with message passing.

Table 6: Random read latency in µs of SPDK, io_uring, libaio, and pread on an Intel Optane Memory PCIe device.

page size [byte]	latency [µs]			
	SPDK	io_uring	libaio	pread
512	2.7	4.4	5.6	4.9
4,096	6.5	8.3	10.3	8.7
16,384	20.1	22.3	26.8	23.1

However, SPDK is not a panacea. Running our workload using `fio`'s SPDK backend does not improve scan or point lookup performance in comparison with the kernel-based I/O stack (i.e., performance is very similar to entry #5 in Table 2). At the same time, because `fio` uses 4 polling threads for our workload, CPU utilization actually increases from 0.76 to 2.76 cycles/byte. This shows that naively using SPDK can increase CPU load, and that SPDK is not a simple drop-in replacement for the traditional I/O stack.

Nevertheless, there are ways of using SPDK in a beneficial manner. In a read-only microbenchmark (using 16 KB random reads), we were able to read over 7 GB/s, while using only 0.12 cycles/byte. To achieve such good CPU usage results, we have to slow down the SPDK polling frequency. To still exploit full DANA bandwidth, it is then necessary to handle large numbers of I/O operations on every polling cycle. In practice, we can do this by submitting and propagating completed I/O requests up the stack in large batches. We found that, on our system, sleeping times of 50 µs between polling cycles, an I/O depth of 1024, and batch sizes of 256 pages allow us to fully exploit our DANA with very low CPU overhead. Obviously, this comes at the cost of very high per-request latency. Thus, SPDK offers the potential of lowering CPU utilization using strategically-placed sleep calls.

Ignoring CPU utilization, SPDK's polling-based approach can theoretically also be used to improve latency. However, because the latency of standard TLC NAND flash is quite high (around 100 µs), SPDK's latency advantage is only visible with high-cost, low-latency NVMe devices. We therefore measured the random read latency of an "Intel Optane Memory 16GB PCIe M.2" card, which is based on persistent memory rather than NAND flash. Table 6 shows that for small page sizes, SPDK offers up to 2 times lower latencies than the kernel. However, for most NAND-based devices, this benefit is negligible.

To summarize, SPDK's polling-based approach shows potential for either reducing CPU utilization or latency. However, how to achieve both at the same time in a complex mixed workload is still an open question, which we defer to future work.

4.8 SLC Cache and Garbage Collection

SSDs require complex logic to handle the physical characteristics of flash memory (e.g., erase-before-write and wear out). Semantically, this logic is hidden from the user by the flash translation layer (FTL), but its performance effects are observable.

The experiments in Table 2 were run for short periods of time to limit the influence of garbage collection. However, by running the workload for longer durations, we can observe different behavior. In particular, to absorb short write bursts, our Samsung SSD employs a dynamically-managed Single-Level Cell (SLC) write cache. Samsung's marketing term for this is *TurboWrite*, which is specified to be in the range of 6 to 42 GB for our 1 TB SSDs.

To show the impact of SLC, we implemented a similar workload to entry #5 of Table 2 in a micro-benchmark and ran it with per-

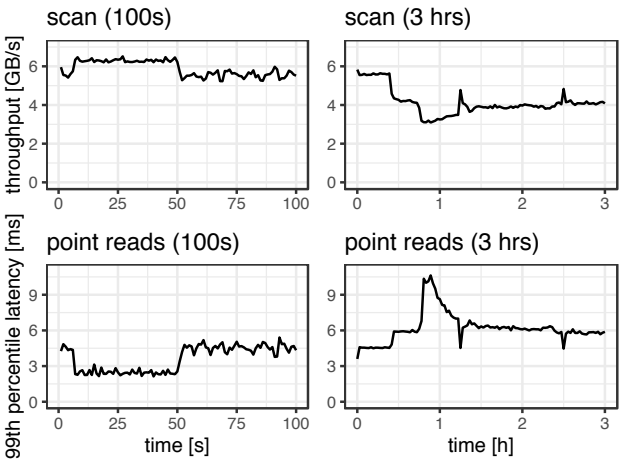


Figure 4: Impact of SLC write cache and GC on throughput and latency over time.

second statistics. For this experiment, we fill our RAID 0 array to 90% of its capacity. The benchmark then uses this range for reads and writes, and the remaining 10% of space are kept free (over-provisioning). The results on the left-hand side of Figure 4 show a sudden performance drop in scan throughput and an increase in tail read latency after 50 seconds. With a write speed of 500 MB/s, we can calculate the amount of dynamically-allocated SLC cache to be less than 25 GB for our four SSDs (≈6 GB each).

An even larger performance drop can be observed when the experiment is run for several hours. On the right-hand side of Figure 4 we can see that after about 30 to 40 minutes, the SSDs are apparently running out of over-provisioned space. This triggers garbage collection, multiplying read latency and halving scan performance. After a while, the SSDs manage to partially recover, but still continues with degraded performance. These effects are, of course, very dependent on the specific SSD model. In Appendix A we execute long-running experiments on enterprise SSDs and find that performance is much more stable on them.

Finally, let us note that SSDs without proper cooling are also prone to performance issues caused by thermal throttling. This is not an issue for our setup, as the PCIe adapter card uses a fan to keep the SSDs cool. We observed severe performance degradations in SSD setups without fans or heat sinks.

5. IMPLICATIONS FOR DBMS DESIGN

In terms of its software interface, flash is quite similar to disk. Both rely on block-wise access through system calls. Therefore, from a software perspective, flash may just seem like a much faster version of disk. Given that main-memory database systems cannot exploit flash, one may wonder whether the traditional disk-based database system architecture is suitable for DANA, or whether new techniques are required. To answer this question, let us compare PostgreSQL, a traditional disk-based system, with LeanStore, a modern storage engine design.

5.1 PostgreSQL on DANA

To evaluate PostgreSQL, we use the TPC-C implementation of OLTP-Bench [5]. As mentioned earlier, PostgreSQL performs I/O using traditional blocking system calls and relies on OS buffering. This corresponds to the slow baseline configuration in entry #1 of Table 2. We use 40 worker threads, 300 warehouses (about 30 GB of data), software RAID 0, and the XFS file system. We compare

Table 7: TPC-C performance with 40 worker threads and 300 warehouses (30 GB data).

	buffer pool GB	perf. ktxn/s	read GB/s	write GB/s
PostgreSQL	16	9	0.2	0.07
	50	10	0.1	0.05
LeanStore	16	130	0.9	1.4
	50	410	0	0.8

the in-memory performance with an out-of-memory scenario where only half the data fits into RAM. For the out-of-memory configuration we restrict the available system memory to 16GB.

Surprisingly, as Table 7 shows, with PostgreSQL the performance drop for going out-of-memory is fairly small. The peak performance only drops by about 10% and only entails a slight increase of I/O usage. This is caused by the fact that PostgreSQL is effectively CPU bound even in this setting. The I/O statistics show that, because of its inefficient CPU use, PostgreSQL is simply too slow to be able to exploit the full potential of our high-performance DANA.

These results indicate that traditional disk-based systems are not a good fit for DANA, despite the fact that flash looks like a drop-in replacement for disk.

5.2 LeanStore on DANA

The fact that disk-based systems are too slow and in-memory systems are fundamentally unsuitable for exploiting DANA, means that a new class of database systems is necessary. A number of high-performance storage managers have recently been proposed [1, 8, 10, 16]. What these systems have in common is that they utilize both DRAM and high-performance storage devices. Such caching-based approaches are more economical than relying on a single memory/storage technology [11].

Compared to PostgreSQL, LeanStore [10] achieves 41× higher in-memory performance, and is still 14× faster for the out-of-memory configuration. In the in-memory experiment only the WAL is written to flash, and the out-of-memory setting results in WAL writes, background writes, and latency-critical reads. The results in Table 7 indicate that LeanStore is currently well optimized for the usage of a single NVMe SSD, and is showing promising results for DANA. We ran LeanStore in a configuration similar to #4 of Table 2, where it avoids `fsyncs` on the critical path, uses `O_DIRECT`, and relies on asynchronous I/O with libaio.

5.3 Recommendations

Based on the experimental findings, let us state a number of recommendations for DANA storage engines:

- Flush operations like `fdatasync` must be moved off the critical path by relying on persistent memory or NVDIMM for logging.
- For scans, asynchronous I/O operations should be used.
- The OS buffer cache incurs a large CPU overhead and needs to be replaced by lightweight user-space buffering.
- The database has to be stored on a scalable file system like XFS, or directly on the block device.

6. SUMMARY AND FUTURE WORK

Arrays of 4, 8, or even 16 PCIe-attached solid-state drives offer high capacity at low prices as well as a bandwidth that is

soon to be on par with DRAM. Although these properties make a *Directly-Attached NVMe Array (DANA)* highly attractive for scan-oriented OLAP workloads, exploiting the high bandwidth presents major technical challenges. We found that disk-based database systems using the traditional I/O stack are not capable of utilizing a DANA—mainly due to high CPU load. The high performance of a DANA requires rethinking the way I/O operations are performed through all database and operating system layers.

In the future, we will work on making LeanStore's internal I/O operations more scalable, optimize RAID for flash, and investigate whether SPDK can fully replace the kernel's I/O stack.

7. REFERENCES

[1] J. Arulraj, A. Pavlo, and K. T. Malladi. Multi-tier buffer management and storage system design for non-volatile memory. *CoRR*, 2019.

[2] M. Athanassoulis, A. Ailamaki, S. Chen, P. B. Gibbons, and R. Stoica. Flash in a DBMS: where and how? *IEEE Data Eng. Bull.*, 33(4), 2010.

[3] M. Bjørling, J. Gonzalez, and P. Bonnet. LightNVM: The Linux Open-Channel SSD subsystem. In *FAST*, 2017.

[4] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD bufferpool extensions for database systems. *PVLDB*, 3(2), 2010.

[5] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[6] J. Do, D. B. Lomet, and I. L. Picoli. Improving CPU I/O performance via SSD controller FTL support for batched writes. In *DaMoN*, 2019.

[7] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging DBMS buffer pool using SSDs. In *SIGMOD*, 2011.

[8] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *SIGMOD*, 2015.

[9] I. Koltsidas and S. Viglas. Flashing up the storage layer. *PVLDB*, 1(1), 2008.

[10] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. LeanStore: In-memory data management beyond main memory. In *ICDE*, 2018.

[11] D. B. Lomet. Cost/performance in modern data stores: how data caching systems succeed. In *DaMoN*, 2018.

[12] Y. Lv, B. Cui, B. He, and X. Chen. Operation-aware buffer management in flash-based systems. In *SIGMOD*, 2011.

[13] S. T. On, Y. Li, B. He, M. Wu, Q. Luo, and J. Xu. FD-buffer: a buffer manager for databases on flash disks. In *CIKM*, 2010.

[14] I. Petrov, A. Koch, S. Hardock, T. Vinçon, and C. Riegger. Native storage techniques for data management. In *ICDE*, 2019.

[15] I. L. Picoli, P. Bonnet, and P. Tözün. LSM management on computational storage. In *DaMoN*, 2019.

[16] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato. Managing non-volatile memory in database systems. In *SIGMOD*, 2018.

[17] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. Persistent memory I/O primitives. In *DaMoN*, 2019.

[18] T. Wang, R. Johnson, and I. Pandis. Query fresh: Log shipping on steroids. *PVLDB*, 11(4), 2017.

APPENDIX

A. ENTERPRISE SSDS

So far, all experiments in this paper were run on consumer hardware. However, the Samsung consumer SSDs have shown sub-optimal performance characteristics on HTAP workloads (Section 4.8). In this section, we therefore run some additional benchmarks on a server system with enterprise SSDs and compare the results.

For the following experiments we use four 1.6 TB Huawei ES3600P V5 SSDs with specified 3,500 MB/s read and 1,900 MB/s write throughput. The SSDs are connected over U.2 to a server with a 64-Core AMD EPYC 7702P processor. In the following, we will denote this as the *enterprise* setup. The *consumer*-grade setup will remain the same as described in Section 3 with four 1 TB Samsung 970 Evo Plus SSDs.

A.1 Microbenchmarks

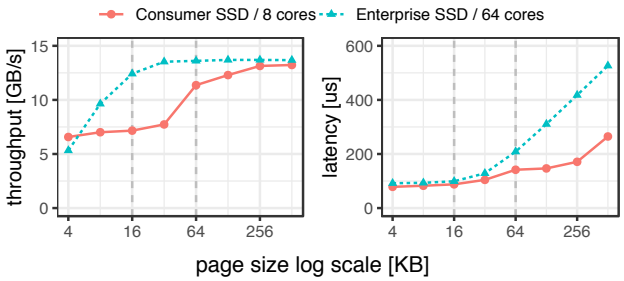


Figure 5: Impact of page size on random reads.

First, we run some basic benchmarks to determine the key differences between the SSDs. Figure 6 compares different aspects of the impact of page size for random reads on the two systems. The most interesting finding here is that our enterprise SSD can achieve close to peak bandwidth with a smaller page size. While the consumer SSD only achieves around half of its specified bandwidth at 16 KB page size, the enterprise SSD already comes close to 90% of its peak bandwidth. Synchronous (queue depth 1) read latency is slightly higher on the enterprise SSD than on the consumer SSD as specified.

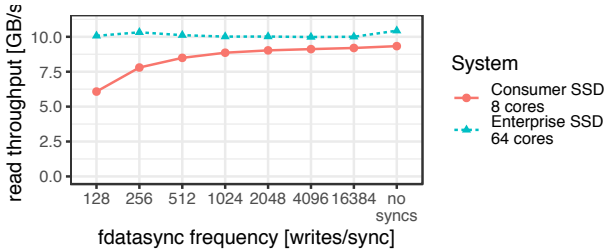


Figure 6: Read throughput on HTAP workload relative to the frequency of `fdatasync` operations.

Another big difference between the two SSDs is the latency of flush operations. Depending on whether the SLC cache has been exhausted or not, we measured the latency for `fdatasync` to be between 1 ms to 3 ms on the consumer SSD. We found that flush operations completely block all other I/O requests until the flush is fully processed. This essentially means the whole SSD is blocked for multiple milliseconds for every flush. The latency of enterprise SSDs is orders of magnitude lower at around 20 μs. Such a low latency can only be achieved by battery backing the DRAM write

Table 8: Results for the HTAP workload as in Table 2 on the *consumer* setup. #1, #2, and #3 are identical. #4 and #5 run with four scan threads.

RAID 0 / file system / OS buffer / synchronous / fsync

#	description	Section	I/O interface	scan MB/s	point MB/s	bg. wr. MB/s	WAL MB/s	total I/O MB/s	CPU cycles/byte
1	baseline	4.1	pread/pwrite/sync	3,503	27	250	6	3,786	6.92
2	− `fdatasync`	4.2	pread/pwrite	5,055	39	250	250	5,595	10.52
3	+ async. I/O	4.3	io_uring	3,439	118	252	251	4,059	7.61
4	+ `O_DIRECT`	4.3	libaio	6,527	20	252	251	7,049	1.43
5	− file system	4.4	libaio	6,685	17	249	249	7,200	1.03

Table 9: Results for the HTAP workload as in Table 2 on the *enterprise* setup.

RAID 0 / file system / OS buffer / synchronous / fsync

#	description	Section	I/O interface	scan MB/s	point MB/s	bg. wr. MB/s	WAL MB/s	total I/O MB/s	CPU cycles/byte
1	baseline	4.1	pread/pwrite/sync	1,513	12	6	76	1,608	228.60
2	− `fdatasync`	4.2	pread/pwrite	1,474	12	13	250	1,749	209.23
3	+ async. I/O	4.3	io_uring	565	16	37	159	778	262.26
4	+ `O_DIRECT`	4.3	libaio	5,445	77	250	249	6,021	1.71
5	− file system	4.4	libaio	9,806	27	246	250	10,328	1.03

cache on the SSD to ensure persistency on power outage. Figure 6 shows the relation between the number of `fdatasync` operations, e.g., one `fdataync` for every 128 writes, and simultaneously running scans. The consumer SSD throughput is clearly degraded by frequent `fdatasync` operations, but its impact gets smaller the rarer they occur. On the enterprise SSD there is no apparent effect.

A.2 HTAP Workload

Table 8 and Table 9 show the result for the same HTAP workload as in Table 2. The difference is that here we use four scan threads instead of one to achieve higher throughput. This is beneficial for our server, because the 64 core CPU is running at a lower clock rate and would not achieve full I/O bandwidth with a single thread. With four scan threads and high queue depth both SSD arrays are fully saturated. Table 8 shows that the consumer SSD was also not fully utilized by the single scan thread.

Just moving the HTAP workload to an enterprise server setup does not solve the issues. Table 9 shows that it actually makes it worse for configurations #1, #2, and #3. The issue here is not the SSD, but rather the higher synchronization overhead in the page cache on the server CPU. Configuration #4 still being slightly slower might also be caused by synchronization in the file system. Finally, with #5 we get a very good result for throughput, close to the peak bandwidth of the SSDs.

Lastly, we run the HTAP workload in configuration #5 for longer periods on both systems. Figure 7 and Figure 8 show the results. The enterprise setup has very stable performance and there is no apparent sign of GC activity, probably due to large internal flash over-provisioning.

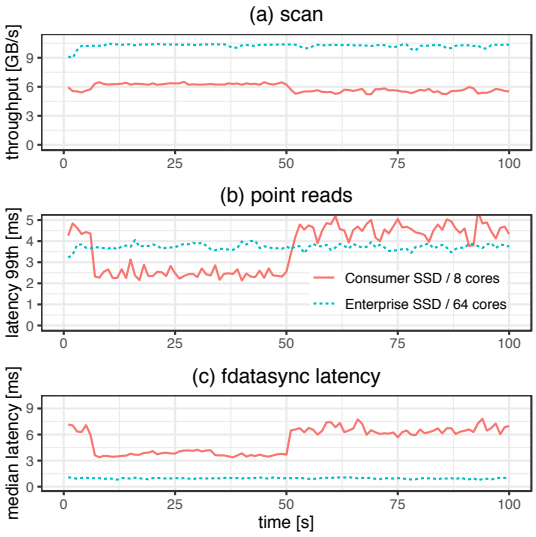


Figure 7: Impact of SLC write caching over time.

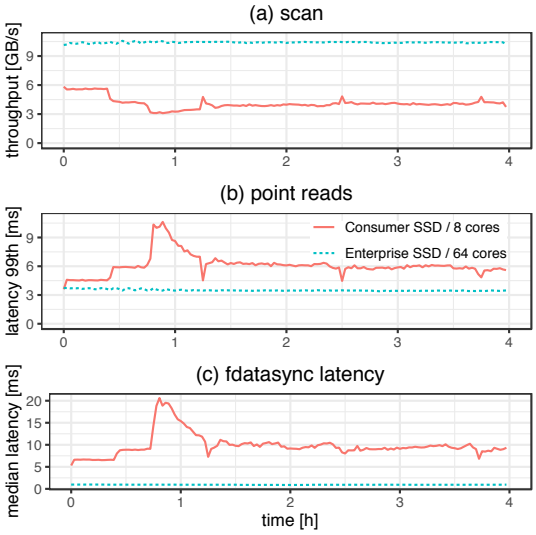


Figure 8: Impact of GC over time.