What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage Engines

Gabriel Haas
Technische Universität München
gabriel.haas@tum.de

Viktor Leis
Technische Universität München
leis@in.tum.de

ABSTRACT

NVMe SSDs based on flash are cheap and offer high throughput. Combining several of these devices into a single server enables 10 million I/O operations per second or more. Our experiments show that existing out-of-memory database systems and storage engines achieve only a fraction of this performance. In this work, we demonstrate that it is possible to close the performance gap between hardware and software through an I/O optimized storage engine design. In a heavy out-of-memory setting, where the dataset is 10 times larger than main memory, our system can achieve more than 1 million TPC-C transactions per second.

PVLDB Reference Format:
Gabriel Haas and Viktor Leis. What Modern NVMe Storage Can Do, And How To Exploit It:
High-Performance I/O for High-Performance Storage Engines. PVLDB, 16(9): 2090 - 2102, 2023.
doi:10.14778/3598581.3598584

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/leanstore/leanstore/tree/io.

1 INTRODUCTION

Flash performance. In the past decade, flash SSDs have displaced magnetic disks as the default persistent storage medium for operational database systems. More recently, the SATA interface has been replaced by PCIe/NVMe, which unlocked previously unprecedented storage throughput: using four PCIe 4.0 lanes, a single SSD can achieve more than one million random I/O Operations Per Second (IOPS) and a bandwidth of 7 GB/s. Because modern commodity servers have up to 128 PCIe lanes per socket, a single-socket server can easily host 8 (or more) SSDs at full bandwidth. The trend of ever-increasing storage bandwidths will continue: servers with PCIe 5.0 are already available and corresponding SSDs with 12 GB/s per device have been announced. This means that arrays of NVMe SSDs are approaching DRAM bandwidth.

Flash capacity. SSDs not only have high throughput, they are also cheap: after a decade of stagnating DRAM prices and rapidly decreasing flash prices, enterprise-grade SSDs cost less than $200 per TB, which is about 10-50 times cheaper than DRAM. We can illustrate this through an example where we have a total server budget of $15,000. For half that budget, one can configure a reasonable server, e.g., 64 core CPU, 512 GB of RAM, and fast networking. The remaining budget can be invested into additional DDR4 RAM modules or PCIe 4.0 NVMe SSDs, resulting in the following two configurations:

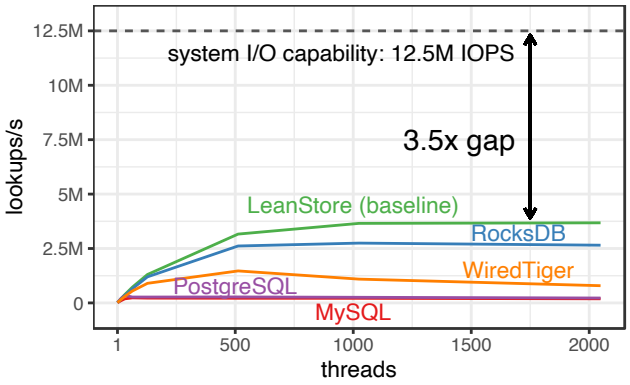


Figure 1: Out-of-memory performance for random lookups in 100 GB database with 10 GB buffer pool and 8 enterprise-grade SSDs.

	RAM		SSD	
	capacity	bandwidth	capacity	bandwidth
config RAM	2.5 TB	150 GB/s	–	–
config SSD	0.5 TB	150 GB/s	8×4 TB	56 GB/s

These configurations illustrate that spending more money on SSDs rather than RAM results in a much larger capacity (32 TB instead of 2.5 TB). This observation has also been made by public cloud providers: AWS offers instances with 8 (i3en) and Azure with 10 (Lsv2) NVMe flash SSDs. Given that alternative storage technologies such as Optane did not achieve commercial success and were consequently discontinued, we believe that flash will remain the only viable option for storing large datasets in a cost-efficient manner.

Performance gap of existing systems. In terms of their basic architecture, DBMSs for flash are similar to disk-based designs: they rely on a buffer pool cache, page-based storage, and B-trees or LSM-trees for indexing. Several modern storage engines, including RocksDB [36] and our LeanStore [1, 26] system, explicitly claim to be optimized for flash storage. However, as Figure 1 shows, existing out-of-memory systems are not capable of exploiting the performance of modern NVMe arrays. In the experiment, we measure the performance of 5 systems for a simple random read benchmark on a 64-core AMD server with 8 Samsung PM1733 SSDs. According

Proceedings of the VLDB Endowment, Vol. 16, No. 9 ISSN 2150-8097.
doi:10.14778/3598581.3598584

to their specification, each of these SSDs can perform 1.5 M random 4 KB reads per second, and consequently one would expect to achieve 8×1.5M=12M lookups/s for this simple workload. In fact, we find that the best system only achieves 3.6 M lookups/s – a 3.5× gap. For the more complex and write-intensive TPC-C, we find an even bigger performance gap of 4.7× between what modern NVMe drives can do, and what existing systems achieve. As we show in this paper, fully exploiting flash storage requires carefully co-designing the storage engine with the flash NVMe SSDs.

Research questions and paper outline. Our high-level goal of closing this performance gap can be broken down into the following research questions:

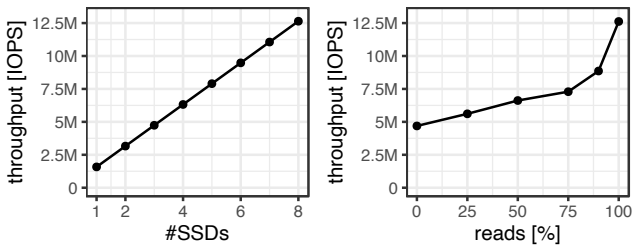
- Q1: Can arrays of NVMe SSDs achieve the performance promised in the hardware specifications?
- Q2: Which I/O API (pread/pwrite, libaio, io_uring) should be used? Is it necessary to rely on kernel-bypassing (SPDK)?
- Q3: Which page size should storage engines use to get good performance while minimizing I/O amplification?
- Q4: How to manage the parallelism required for high SSD throughput?
- Q5: How to make a storage engine fast enough to be able to manage tens of millions of IOPS?
- Q6: Should I/O be performed by dedicated I/O threads or by each worker thread?

To answer these questions, we first perform an experimental study of the hardware characteristics of NVMe flash storage in Section 2. Based on the findings, we derive implications for high-performance storage engines. Section 3 then presents a blueprint for an I/O backend that can fully exploit NVMe arrays.

System integration and techniques. We integrated the techniques proposed in this paper into LeanStore, an open source storage engine prototype. LeanStore was designed with SSDs in mind and already applies several important I/O-related optimizations recommended in earlier work [13], including a fast buffer pool, file system bypassing using `O_DIRECT`, and `fsync` batching. This baseline version is the starting point of this paper and is sufficient to exploit a single NVMe SSD, but, as Figure 1 shows, not 8 of them.

Unprecedented performance. The new version of LeanStore is the first system that is able to fully close the gap. Admittedly our techniques are more carefully engineered than totally new, yet together they result in a practical system design with few configuration parameters that achieves unprecedented out-of-memory performance. We also believe the findings and techniques in this paper are applicable to most other storage engines and database systems.

Results. We evaluate our design in Section 4. On a server with 64 cores and 8 SSDs, the resulting system indeed achieves 12.5 M lookups per second for the random lookup workload. On the much more challenging TPC-C benchmark with a 400 GB buffer pool and a 4 TB dataset, LeanStore executes over 1 M TPC-C transactions per second. For a 20 TB dataset, we still achieve 0.4 M transactions/s. For all workloads, the machine becomes I/O bound if there are enough concurrent user requests – as one would expect for workloads where the dataset is much larger than the buffer pool.



(a) drive scalability **(b) read/write mix**

Figure 2: Drive scalability for random reads and impact of read/write mix (SPDK, 8 SSDs, 4 KB pages, freshly initialized with 10 TB data, 60 second runs).

2 WHAT MODERN NVME STORAGE CAN DO

In this section, we present the result of a series of micro-benchmarks that provide us with the necessary background on how to fully exploit modern storage. All experiments were performed on a 64-core AMD Zen 4 server with 8 × 3.8 TB Samsung PM1733 SSDs. Details on the experimental hardware and software setup can be found in Section 4.1.

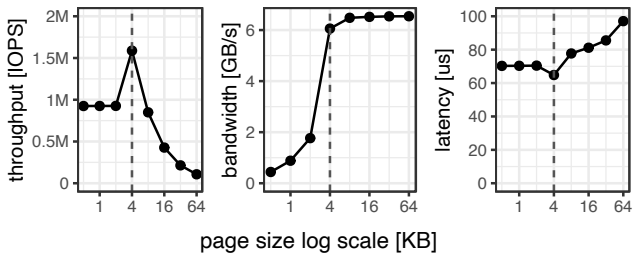
2.1 Drive Scalability

Drive scalability. According to the hardware specification sheet, our Samsung PM1733 SSDs are capable of performing 1.5 M I/O operations per second (IOPS) for random 4 KB reads. Hence, with 8 drives, we should achieve the remarkable number of 12 M IOPS. Figure 2 shows that the throughput indeed scales perfectly with the number of drives used. We actually achieve slightly more than expected, namely 12.5 M IOPS or 1.56 M IOPS per SSD.

Read/write mix. Transactional workloads are often write-intensive, and it is well known that SSDs have asymmetric read/write speed. As Figure 2b shows, with random writes on an empty SSD our hardware setup achieves 4.7 M IOPS. Note that SSD write performance depends on the internal state of the SSD and the write duration. In the worst case of a full SSD and prolonged writing, the throughput will be lower; the data sheet specifies the worst-case, per-drive random write throughput at 135 k IOPS. For OLTP systems, mixed read/write workloads are more common. As Figure 2b shows, with 10% (25%) writes we measured 8.9 M (7.0 M) IOPS. These micro-benchmarks show that modern NVMe storage provides an excellent foundation for transactional systems, which often require many random I/O operations.

2.2 The Case for 4 KB Pages

Page size tradeoffs. In contrast to byte-addressable persistent memory, access to flash happens at page granularity. Many database systems use larger pages sizes such as 8 KB (PostgreSQL, SQL Server, Shore-MT), 16 KB (MySQL, LeanStore), or even 32 KB (WiredTiger) as their default. In LeanStore, we observed that for in-memory workloads, larger page sizes generally improve performance, which is why we originally chose 16 KB [26]. A second benefit of a larger page size is that it reduces the number of distinct entries in the buffer pool and therefore leads to less cache management overhead.



(a) IOPS (b) throughput (c) latency

Figure 3: Impact of page size on IOPS, bandwidth, and latency for random reads (SPDK).

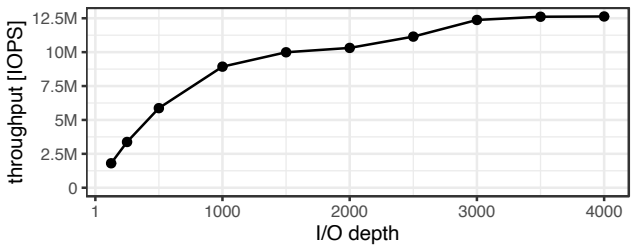


Figure 4: Random read throughput with 4 KB pages depending on number of outstanding I/Os (SPDK).

However, the big downside of larger pages is I/O amplification on out-of-memory workloads. With 16 KB pages, for example, reading or writing 100 Byte records results in an I/O amplification of 160×. A smaller page size of 4 KB would reduce amplification by 4×.

Better go small (but not too small). For data center grade SSDs, we found that the sweet spot for the page size is 4 KB, as it allows for the highest random-read performance and lowest latency. Figure 3 shows that it is possible to achieve almost the full bandwidth (6 GB/s) with 4 KB pages and random reads. This is only 8% slower than the maximum bandwidth of 6.5 GB/s that can be achieved with larger pages (or sequential access). Looking at the latency for different page sizes in Figure 3, one can observe that the latency generally increases with page size and has its minimum at 4 KB. Technically, NVMe allows even smaller pages – down to 512 bytes. For write amplification this would be even better, but our results show that using smaller pages than 4 KB significantly decreases performance[1]. In fact, a smaller page size actually results in worse IOPS and latency on our SSDs, as can be seen in Figure 3. This is due to higher overhead in the flash translation layer, together with the fact that the flash hardware internally is not optimized for 512 byte pages[2]. We argue that the gains from lower latency and I/O amplification make 4 KB pages the best choice for systems that optimize for out-of-memory performance. However, to benefit from such a small page size in terms of performance, the DBMS must be able to deal with the resulting high buffer pool and I/O management work. In fact, most DBMS will not benefit *just* from using smaller pages because I/O throughput is not the limiting factor (c.f., Section 4.2).

2.3 SSD Parallelism

SSD parallelism. Internally, SSDs are highly parallel devices, with multiple channels connecting to independent flash dies. Getting enough I/O requests to the SSD can be difficult, as it requires a large number of simultaneous requests for high performance. Flash random read latency is on the order of 100 microseconds, which is 100× faster than disk, but still 1000× slower than DRAM. With synchronous accesses (i.e., sending a new I/O request only after the previous one is completed) this would result in a meager 10k IOPS (or 40 MB/s). Thus, to get good throughput from SSDs, one has to exploit their internal parallelism by asynchronously scheduling many concurrent I/O requests. Figure 4 shows the relation between IO-depth, i.e., the number of simultaneously outstanding I/O requests on all 8 SSDs, and overall throughput. We can see that around 1000 concurrent I/O requests, i.e., more than 100 per device, are necessary to get decent performance, and 3000 to fully saturate the system. One of the main challenges for a database system exploiting NVMe arrays is managing these high numbers of I/O requests.

2.4 I/O Interfaces

Modern operating systems offer several interfaces for storage I/O. Here, we discuss the most common I/O libraries on Linux. In the end, all interfaces end up doing the same for NVMe drives: take the I/O requests and place them into the NVMe submission queues that are used by the host system to communicate with SSDs. When the SSD has completed any requests, it will write completion events to the completion queue and, if interrupts are used, notify the host. The libraries differ in how requests are submitted and how completed operations are reaped.

Blocking POSIX interface. The classic and most common way of doing I/O on Linux is by using POSIX system calls like `pread` and `pwrite`. POSIX calls for file operations are usually used in a blocking fashion, where a single I/O request is submitted at a time. The kernel will then block the user thread until the request is handled by the drive. This is shown in Figure 5a.

libaio: traditional asynchronous interface. libaio is an asynchronous I/O interface that allows the submission of multiple requests with one system call. This saves context switches between user and kernel mode, and allows a single thread to perform multiple I/O operations simultaneously. As Figure 5b illustrates, I/O requests are submitted in a non-blocking fashion through `io_submit()`, which immediately returns, and the program has to poll for completions with `get_events()`.

io_uring: modern asynchronous interface. io_uring [38] is the designated successor to libaio. io_uring implements a new generic asynchronous interface to the kernel that can be used for storage, but also for networking. As Figure 5c shows, it is based on shared queues between kernel and user-space. Multiple requests can be added to the submission queue, and with a single `io_uring_enter()` system call the kernel can be notified of the available requests. Inside the kernel, an I/O request will essentially go through the same abstraction layers (file system, cache, block device layer, etc.) as

[1]This is quite different from Optane SSDs, which are based on Phase Change Memory rather than flash, and therefore offer lower latencies with very small pages sizes [40].
[2]Internally, SSDs have larger physical register sizes.

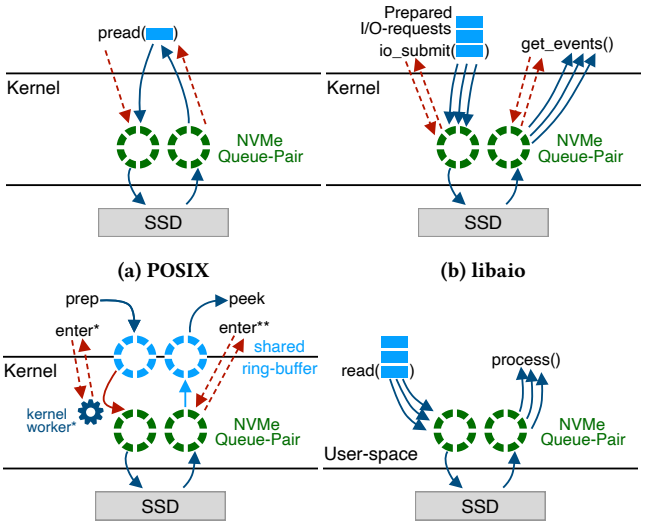


Figure 5: Comparison of Linux storage I/O interfaces.

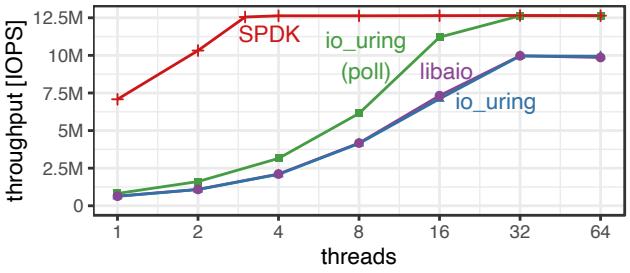


Figure 6: Random read throughput (measured with our custom I/O benchmarking tool, without request batching).

with the other kernel interfaces. After going through all these kernel layers, the request will end up in an NVMe submission queue. Linux also implements a submission queue polling mode (`SQPOLL`) where the kernel spawns kernel-worker threads (marked with * in the figure) to poll the submission queue. In this mode, at the cost of additional kernel worker threads, no system calls are required.

Polling I/O completions. For all interfaces discussed so far, the default way of notifying the host about completed I/O events is hardware interrupts. With io_uring, it is possible to disable interrupts (`IOPOLL`). Using this mode, the application must poll completion events (marked with ** in the figure) from the completion queue. Avoiding interrupts can reduce latency and CPU overhead in situations with high IOPS. With I/O polling, the `io_uring_enter()` system call is also used to poll on the NVMe completion queue. When `SQPOLL` is set, I/O polling is handled by a kernel worker thread and completions can be reaped directly from user space without requiring a system call.

User-space I/O with SPDK. Intel's Storage Performance Development Kit (SPDK) is a set of libraries and tools used for high-performance storage applications [39]. Specifically, the SPDK NVMe driver, which is the user-space driver for NVMe SSDs, is relevant for us. As Figure 5d illustrates, to communicate with an NVMe SSD, SPDK directly allocates the NVMe queue pairs (for submission and completion) in the user space. Submitting I/O requests is therefore as simple as writing a request into a ring buffer in memory and notifying the SSD that there are new requests available through another write. SPDK does not support interrupt-driven I/O and completions always have to be polled from the NVMe completion queue. SPDK completely bypasses the operating system kernel, including the block device layer, file systems, and the page cache.

2.5 A Tight CPU Budget

Available CPU cycles. The experiments in Section 2.1 show that modern storage hardware is capable of achieving tens of millions of IOPS. Managing that many requests becomes an important task for the database system and can take up a significant amount of CPU time. To get a feeling about the available CPU budget for handling 12 M I/O operations per second, consider the following back-of-the-envelope calculation: Using the specification for our AMD CPU we get a CPU budget of 13k cycles per I/O operation (2.5 GHz × 64 cores / 12 M IOPS).

CPU impact of I/O libraries. In the microbenchmark shown in Figure 6, we measured the throughput achievable with a varying number of threads. The figure shows that with libaio and io_uring (interrupt based), the full bandwidth cannot be reached. With 32 threads, the maximum throughput is 10 M IOPS, with most of the time being spent in the kernel. With io_uring in poll mode, the results are better: with 16 threads one can get close to the full throughout. Note that to achieve these numbers we had to disable most operating system features, such as the file system, RAID, and the OS page cache [13]. In the microbenchmark, SPDK achieves far better results and is capable of reaching the full bandwidth with only three threads. However, the use of user-space I/O has major implications for users and on the design of the I/O backend, which we will discuss later in the paper.

fio can be the bottleneck. The experiments shown in Figure 6 were measured with a custom benchmarking tool. We also executed the same benchmarks with the standard I/O benchmarking tool fio [2] and, surprisingly, got worse results. In particular, fio's implementation of interrupt-based I/O cannot achieve the full bandwidth with random 4 KB reads. The fact that a specialized I/O benchmarking tool can become a bottleneck indicates that achieving high I/O throughput in full-blown database systems is difficult.

Every cycle counts. Our experiments show that, unless we use SPDK (which is not always a practical choice), around half of the available CPU cores are consumed by the OS just for submitting and reaping I/O requests. Coming back to our back-of-the-envelope calculation, this leaves us with only 6500 cycles for all remaining DBMS work. This includes query processing work, index traversal, concurrency control, logging, buffer manager overhead, page eviction, and scheduling. Note that our calculation assumes perfect multi-core scalability for all DBMS components, i.e., the budget shrinks if scalability is less than perfect.

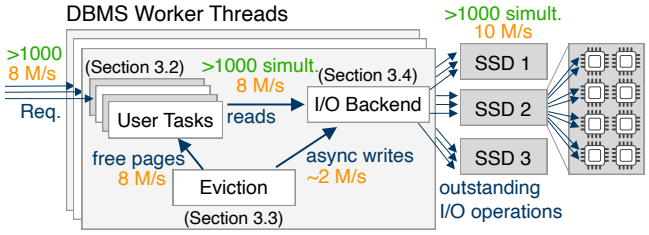


Figure 7: Design overview illustrating request-level, CPU-level, and SSD-level parallelism.

2.6 Implications for High-Performance Storage Engines

The gap. Our high-performance storage engine LeanStore has specifically been designed for NVMe SSDs. In contrast to older systems, it is not CPU bound and scales well on multi-core CPUs. However, as Figure 1 shows, it achieves only 3.6 M IOPS in a situation where 12.5 M IOPS should be possible. The experiments in this section allow us to pinpoint where this 3.5× gap comes from.

Not enough parallelism. LeanStore differentiates between page-provider and worker threads. Page-provider threads are responsible for the eviction strategy and for writing dirty pages. To achieve decent performance in heavy out-of-memory workloads, multiple page-provider threads are required to find enough cold pages for eviction. Worker threads are responsible for handling user queries, with each query running exclusively in its own operating system thread. Since worker threads use synchronous read operations (`pread`) to handle buffer manager page faults, a context switch to the kernel is required with every request. The thread will then be blocked until the I/O request is completed. As we saw, SSDs require many concurrent I/O operations to be saturated – hence, thousands of worker threads are necessary. To achieve the performance shown in Figure 1, the 32 page provider threads had to be isolated from worker threads, to get enough CPU time and not become the bottleneck. Page provider threads are pinned to CPU cores and workers can freely use all other CPU cores as scheduled by the kernel.

Oversubscription problems. This means that thousands of worker threads are now competing over the remaining CPU cores, which leads to very high context switching overhead. After starting 500-1000 threads, all CPU cores are running at 100% load, with most time spent in the kernel. Running the system with such high oversubscription is neither efficient nor robust.

Summary. The experiments in this section demonstrate that modern NVMe storage offers tremendous capabilities. However, getting close to these hardware limits is non-trivial even in microbenchmarks (as evidenced by the fio bottleneck), and is a major challenge in full-blown storage engines. We also found that 4 KB pages offer the best trade-off between random IOPS, throughput, latency, and I/O amplification. However, such a small page size stresses the I/O backend even more.

3 HOW TO EXPLOIT NVME STORAGE: AN I/O-OPTIMIZED STORAGE ENGINE DESIGN

Exploiting NVMe arrays. In this section, we describe a storage engine design that can exploit the full potential of modern storage devices. As our starting point, we use a baseline version of LeanStore. As Figure 1 shows, this version of LeanStore is faster than other systems, but is not capable of fully exploiting NVMe arrays. Note that simply switching to 4 KB pages and polled I/O alone will not achieve much, as one will immediately run into software bottlenecks. High out-of-memory performance requires changes throughout the entire system.

Parallelism is the problem. At a high level, the main challenge we face is the parallelism of modern hardware: First, we have parallelism at the request level, which may either directly come from user queries or from the DBMS itself (e.g., through intra-query parallelism or prefetching). Second, we have dozens or hundreds – but not thousands – of CPU cores of a modern server. Third, we have 1000+ I/O operations outstanding at any point to keep SSDs busy.

3.1 Design Overview and Outline

The solution is to embrace parallelism in every part of the system. Figure 7 shows a high-level view of our system and the parallelism it has to handle. Going from left to right in the figure, we see that everything starts with concurrent requests. To manage all these requests, we employ cooperative multitasking where tasks are scheduled by the DBMS in user-space as described in Section 3.2. All these I/O requests eventually result in a very high demand for free pages (e.g., 8 M/s in the figure). This demand must be satisfied by an efficient page replacement algorithm and asynchronous dirty page writing. In Section 3.3, we argue that these CPU-intensive tasks should be integrated into the worker threads. Finally, the I/O backend connects the DBMS with the storage hardware using one of the I/O libraries introduced earlier. Section 3.4 discusses several models for how to connect worker requests to the hardware. All this work is done in parallel by a limited number of CPU cores. Since the overall CPU budget is limited, every component involved must be heavily optimized and implemented in a scalable fashion. These optimizations are described in Section 3.5.

3.2 DBMS-Managed Multitasking

The problem with high oversubscription. Most database systems run queries in independent threads and use synchronous I/O requests (i.e., `pread`) to handle page faults. In this design, to keep the SSDs busy there must be more than 1000 threads running simultaneously for user requests only. If queries take significant amounts of CPU time fewer I/O requests are generated and even more threads are necessary. Even large servers do not have thousands of cores, which means that a synchronous I/O causes extreme oversubscription and poor performance due to context switching overhead.

Lightweight tasks. To avoid oversubscription, we use lightweight cooperative threads that are managed by the database system in user-space. This reduces the context switching overhead and allows the system to be fully in control of scheduling without kernel interference. In this design, which is illustrated in Figure 8, the system

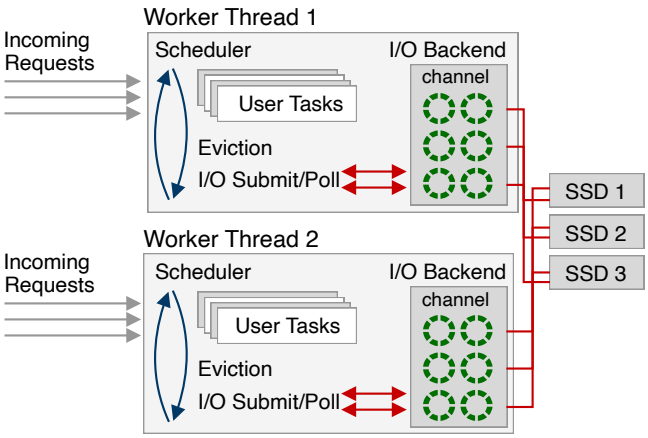


Figure 8: Design overview. The system is handling many incoming requests in parallel. Worker threads are running these requests cooperatively as tasks, while also taking care of page eviction, I/O submission, and polling.

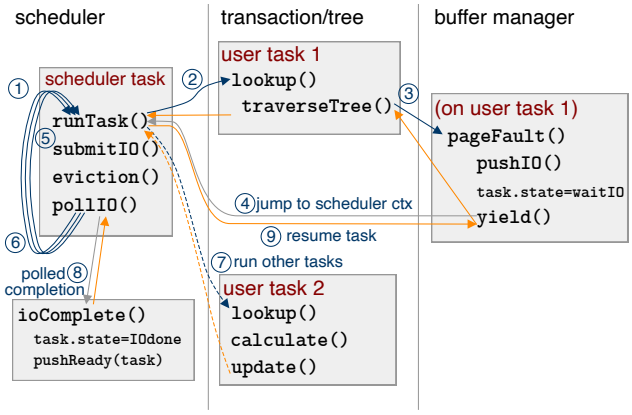


Figure 9: Exemplary sequence of events in a worker thread when handling a page fault in a user task.

starts as many worker threads as there are hardware cores available in the system. Each of these workers runs a DBMS-internal scheduler that executes these lightweight threads, which we call tasks. To implement user-space task switching, we use the Boost context library [22], specifically `fcontext`. Thereby, a task switch costs only around ~20 CPU cycles, instead of several thousand for a kernel context switch. This enables cheap and frequent context switches deep in the call stack, and makes it fairly easy to port existing code bases to this new design.

Cooperative multitasking. Conceptually, Boost contexts are non-preemptive user-space threads. Tasks therefore need to yield control periodically back to the scheduler. In our cooperative multitasking design, this happens whenever a user query encounters a page fault, runs out of free pages, or when the user task is completed. Further, to prevent a worker from being stalled due to latching, we modified all[3] latches to eventually yield to the scheduler as well.

Non-blocking I/O. Using non-preemptive tasks means that blocking I/O requests (i.e., pread) cannot be used, as it would block the whole worker thread. Instead, we switch to a design where worker threads use non-blocking I/O interfaces, such as libaio, io_uring, or SPDK, to submit requests asynchronously. When the task encounters a page fault, the I/O request will be submitted to the I/O backend and the corresponding task will yield execution back to the scheduler. Each worker can therefore have multiple outstanding I/O requests. When an I/O is finished, the worker will (eventually) jump to the corresponding task. This makes running thousands of queries simultaneously as lightweight user-space threads feasible.

3.3 Background Work Through System Tasks

Page eviction is crucial. Database systems are not only running user queries, but also performance-critical tasks such as page eviction. Once the system runs out of free pages because the page replacement algorithm is too slow, the system will stall. The same will happen if dirty pages cannot be written out fast enough.

[3]This includes latches protecting pages as well as internal DBMS data structures.

The burden of background threads. For efficiency reasons, the original LeanStore version evicts and writes pages in batches. This is done by background threads called *page providers* [15]. With millions of page evictions and hence I/O operations per second, multiple such threads are needed. The downside of background threads is that it is hard to know how many of them are needed, in particular when workloads change.

Background work as tasks. We switched to a completely asynchronous and non-blocking model. Since our new approach is already using tasks to run user queries, we run background work as (system) tasks as well. This removes the necessity to have background threads, and page eviction can instead be run directly on worker threads. Another advantage is that worker threads do not stall waiting for free pages due to page providers being too slow. Instead, workers naturally start spending more CPU time on eviction if the system starts running out of free pages. This is implemented by checking whether there are enough free pages on every context switch. As an optimization, eviction is then run directly in the main context (instead of a separate system task).

Example. Figure 9 shows an example illustrating how worker threads execute tasks. Each worker continuously runs a loop ① alternating between running tasks, submitting I/O, page eviction, and polling for I/O completions. When the scheduler decides to run a user task ② it will create the necessary context and pass control to the specific task. In the example, user task 1 is a lookup in the B-Tree. Traversing the tree, the lookup might encounter a node that is not available in the buffer pool. This triggers a page fault ③, in which an I/O request is set up and pushed to the I/O backend. The task state is set to `waitIO` and execution yields back to the scheduler running on the main context. The task remains blocked until the page fault is resolved. Back in the worker thread loop, the scheduler will trigger the submission ⑤ of the I/O request to the SSD[4]. Next, the replacement strategy will check if there are enough free pages available and, if necessary, run page eviction routines. Lastly, the I/O backend will be polled for completions. The worker

[4]Our implementation submits each I/O immediately. One could also implement a batching heuristic that tries to collect several requests and submit them in a batch.

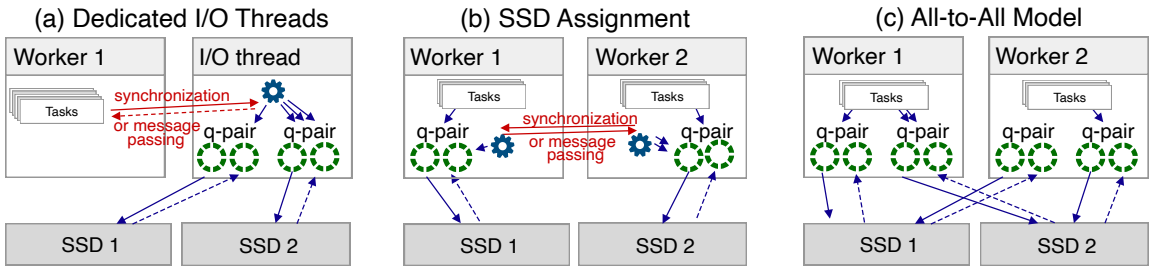


Figure 10: I/O model comparison. We use the all-to-all model, which does not require message passing between threads.

thread will then continuously loop ⑥ through these jobs. Afterwards, user task 2, consisting only of CPU work, is executed ⑦. This task can run to completion as long as it does not encounter any page faults or blocked latches. When the task is finished, the scheduler code will resume, and at some point the previously-submitted I/O request will be completed. The I/O backend will run the callback associated with the request ⑧, set the task state to `IOdone` (i.e., ready) and push the task back into the ready queue. The task can then be resumed ⑨: the page fault is resolved and the tree traversal of user task 1 continues.

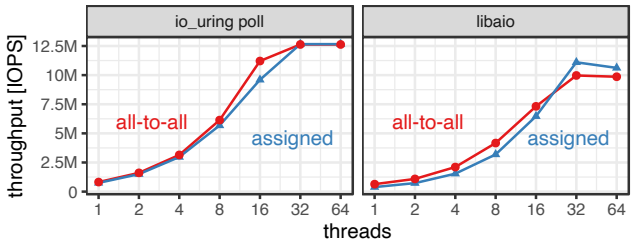


Figure 11: Random read performance of all-to-all and assigned SSD models (random reads, 4 KB pages).

3.4 Managing I/O

Unified I/O abstraction. Our I/O backend supports all major asynchronous I/O libraries (libaio, io_uring, and SPDK) and implements a low-overhead RAID0-like abstraction (data striping) that logically combines multiple SSDs to one logical device. Implementing RAID0 in the DBMS rather than relying on the Linux software RAID implementation improves performance (cf. Figure 13) and enables a uniform interface across all I/O libraries, including SPDK. The backend also abstracts away the details of how to perform I/O operations asynchronously. There are two open questions regarding I/O management: First, should there be dedicated I/O threads that are solely responsible for handling I/O, or should worker threads handle both user tasks and I/O? Second, should a thread be responsible only for handling one SSD or should all threads access all SSDs?

I/O models. In the *Dedicated I/O Threads* model (Figure 10a), worker threads cannot directly access the SSDs, but have to communicate with I/O threads that handle all I/O. This requires some form of message passing between worker threads and I/O threads for every I/O operation. This can be implemented in user-space, or in essence this is what io_uring using `SQPOLL` mode does, just that the I/O threads are kernel workers. The *SSD Assignment* (Figure 10b) model is inspired by I/O microbenchmarks that assign SSDs to specific threads. Again, message passing is needed – in this case, between worker threads. In the *All-to-All* model (Figure 10c), every worker can access every SSD without requiring message passing. This includes both reads from user tasks and writes from the background writer. Models a and b have the advantage that they simplify request batching, require fewer memory-mapped I/O (MMIO) calls, and need fewer polling calls. For kernel I/O libraries, this can result in a reduction in context switching and allows for higher peak efficiency in microbenchmarks. Nevertheless, as we discuss below, we believe that the all-to-all model is the best option.

Problems with dedicated I/O threads. For similar reasons as discussed in Section 3.3, dedicated background threads for I/O have major downsides. One could consider using a single thread for handling all I/O. However, a single thread can only achieve 630k (libaio, io_uring) to 820k (io_uring poll) IOPS. We run experiments with io_uring uisng `SQPOLL` as dedicated I/O threads, but this actually decreased performance and efficiency. This is the case because kernel workers take up CPU cores that could otherwise be used by worker threads. Further adjusting the number of I/O threads is difficult and highly dependent on the workload. Just like page eviction, a better approach is to handle I/O work directly as part of the scheduler loop on our worker threads.

SSD assignment model. Microbenchmark implementations often assign each SSD to one specific thread to improve cache locality and reduce polling. This means that a system with 8 SSDs would also require at least 8 threads. In microbenchmarks, dedicated assignment works well as each thread only generates I/O operations for the assigned SSD. A real system, however, would require message passing between the threads, involving some kind of synchronization. Nevertheless, this downside may be justifiable if it results in significantly better I/O performance. To find out whether that is the case, we implemented the SSD assignment and all-to-all models in a microbenchmark. As Figure 11 shows, both approaches achieve similar performance. We therefore adopt the all-to-all model.

Robust all-to-all communication through I/O channels. The NVMe queue pair abstraction is specifically designed with multiple threads in mind. Every thread can have its own queue pair to every SSD. A similar design can be used with libaio and io_uring. Therefore, our I/O backend implements an *I/O channel* abstraction that encapsulates implementation differences. Every worker has one I/O channel that handles I/O requests to all SSDs, without

synchronizing with other threads. This is an efficient and robust solution, as it does not require any additional message passing or synchronization. All worker threads have the same responsibilities and no thread has special roles. This symmetry also means that LeanStore can be efficiently run on a single CPU core.

3.5 CPU Optimizations and Scalability

CPU can be a bottleneck. Conceptually, the techniques introduced above largely solve the problem of managing I/O. However, because traditionally I/O has always rightly been considered to be slow, the I/O path of most systems is less optimized than in-memory parts of the systems. Once the system becomes capable of scheduling millions of IOPS, the CPU often becomes the performance bottleneck. In the following, we describe a number of optimizations that we had to implement in LeanStore to finally become I/O bound.

Scalability. In the original LeanStore design a single global lock [26] was used to protect in-flight I/O operations and the cooling stage from concurrent access. The LeanStore paper [26] argues that this global lock is not a scalability bottleneck, because it is only required on the cold path and even with fast SSDs, an I/O operation is still much more expensive than the lock acquisition. With modern NVMe SSDs, this global lock quickly becomes the performance bottleneck. To resolve it, the replacement strategy and the I/O manager both get independent locks. To avoid contention on these locks, they are logically partitioned by pageId. Hence, every thread can still access all pages but lock contention is reduced. This also means that multiple threads can be used effectively for page eviction, which drastically improves out-of-memory performance. The numbers measured in Figure 1 for LeanStore already include this optimization, and we use it as our baseline in the evaluation. To really make LeanStore even more scalable, page eviction and I/O path have to be streamlined. Page eviction was adapted to shorten critical sections and remove locking with lock free data structures where necessary. Further, all memory allocations were removed, and hot code paths had to be micro-optimized.

Streamlined eviction. In the original LeanStore version, page eviction was done in two phases. In the first phase, random pages are picked and added to the cooling queue. Pages that are accessed while in the cooling stage are removed from it. In the second phase, pages from the end of the cooling queue are evicted. In both phases it is necessary to access the parent node to update the cooling tag in the swizzled pointer. LeanStore does not employ parent pointers, as this simplifies the latching protocol significantly. Without parent pointers, every access to the parent requires a tree traversal from the root node (`findParent()`). When running the system out of memory with millions of evicted pages per second, this frequent tree traversal takes a significant amount of cycles (>10% of total CPU time). To reduce this overhead, we introduced optimistic parent pointers in nodes that are checked for validity when accessed. With this feature, most `findParent()` calls can be saved and performance increases significantly.

4 EVALUATION

In this section, we first compare the complete design implemented in LeanStore with RocksDB and WiredTiger. In an ablation study that starts with the original (baseline) LeanStore version, we then show how much each of the techniques and optimizations proposed in this paper contribute to overall performance. Then, we shift our focus on data scalability before performing a detailed comparison of the different I/O libraries.

4.1 Experimental Setup

Hardware. The experiments were performed on a Linux (5.19) machine with an AMD EPYC 7713 Milan CPU with 64 cores / 128 threads, which has 128 PCIe 4.0 lanes. The system has 512 GB of DRAM and 8 × Samsung PM1733 SSDs with 3.84 TB each. As we have shown in the introduction, every SSD is capable of achieving 1.5 M I/O operations per second. When using all SSDs simultaneously this performance cannot be reached if the *IOMMU* is enabled in the kernel parameters, hence we have it disabled with `amd_iommu=off`. All experiments are run on all 8 SSDs with our RAID 0 implementation in LeanStore. The SSDs are erased (using `blkdiscard` command) before every experiment, as that is the only state in which performance can easily be compared. It's important to note that performance degrades on full SSDs and with prolonged writing due to internal write amplification. This is caused by the SSD's flash translation layer (FTL) performing garbage collection.

Competitors. To compare our results we choose the popular storage engines RocksDB [36] (v6.15.5) and WiredTiger [30] (v3.2.1). RocksDB is a key-value store based on LSM-trees developed and used by Facebook. Like LeanStore, it is optimized for multi-core CPUs and fast storage. WiredTiger is a key-value store that is used as the default storage engine for MongoDB. It supports both LSM and B-tree indexes. In our experiments we run it using B-trees.

Settings. To focus on I/O handling and prevent concurrency control from becoming the bottleneck, in all systems we disable logging and select the least available isolation level. Unless otherwise noted, we configured all systems to use 4 KB pages. We experimented with numerous configurations in order to identify the optimal setup that would yield the best performance for both engines. For example, we used the ideal number of threads in every experiment (e.g., for RocksDB this was 1024 threads with random lookup and 64 with TPC-C). All storage engines are run without kernel page cache (`O_DIRECT`). For the experiments we use TPC-C and a random lookup benchmark with 8 byte keys and 120 byte payloads. Both benchmarks are implemented in C++ and linked to the storage engines.

4.2 System Comparison

TPC-C. For the first experiment, shown in Figure 12, a 16 GB buffer pool is used for all three storage engines. We run the TPC-C workload with 1000 warehouses, which corresponds to about 160 GB of data. Hence, the dataset is 10× larger than the buffer pool. LeanStore is run using its best-performing configuration with user-space threads and SPDK as I/O backend. LeanStore achieves over a million (1.07 M) TPC-C transactions per second (tps) with 64 threads. The other storage engines can use all 128 hardware threads, and we use the optimal number of worker threads to achieve the best performance. In the TPC-C workload, this was 64 threads for both systems. RocksDB achieves around 10k tps and WiredTiger around 40k tps. TPC-C, a fairly intricate workload with lots of inserts and

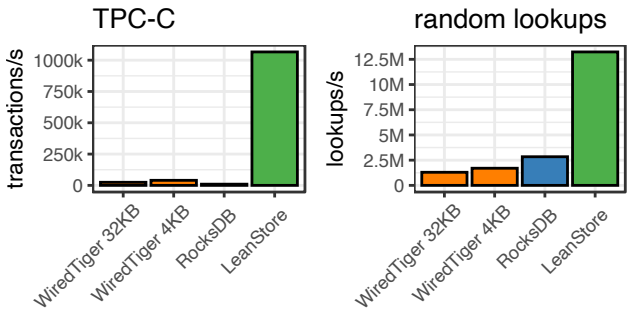


Figure 12: LeanStore vs. competitors (16 GB buffer pool, 160 GB data, LeanStore with SPDK, number of threads chosen for highest performance in random lookups: WiredTiger: 256, RocksDB 1024, LeanStore: 64; with TPC-C: 64 for all storage engines).

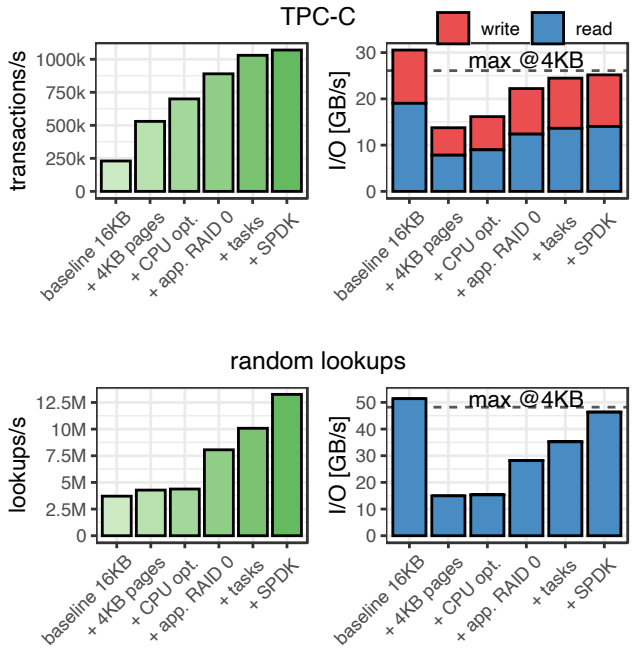


Figure 13: Impact of different features on LeanStore performance and I/O (16 GB buffer pool, 160 GB data size, 4 KB pages (except baseline), libaio (except +SPDK)).

updates, allows LeanStore to deliver orders of magnitude higher performance than the other engines.

Random lookups. The second experiment shows the read-only lookup performance with uniformly-distributed keys. Here, WiredTiger achieves 1.8M lookups per second and RocksDB 2.8M. LeanStore achieves 13.2M lookups/s which corresponds to the full theoretical SSD bandwidth of 12M IOPS, considering 10% of lookups are in-memory.

Page size. In Section 2.2 we argued that 4 KB page size is the optimal page size for fast DBMS using flash SSD arrays. To show that DBMS actually do not benefit a lot by using these relatively small pages, we run WiredTiger with the default 32 KB pages and as comparison with 4 KB pages. With 4 KB pages the TPC-C performance *only* increases from 25k to 40k tps, far away from a theoretically almost 8× higher SSD speed in IOPS. The random lookup workload smaller pages gain even less with *only* 1.3M to 1.7M lookups/s.

4.3 Incremental Performance Improvements

Let us have a look at how much the different optimizations and techniques discussed in Section 3 have increased performance. For this, we are starting from the LeanStore version as described in the original paper as baseline (but add the partitioning optimization for better scalability). Step by step we add additional features: 4 KB pages, CPU optimizations, custom RAID, user-space threads, and SPDK for kernel by-passing. As parameters, we choose again a 10× out-of-memory factor with 16 GB buffer pool and 160 GB of data for both TPC-C and the read-only lookup workload. The results can be seen in Figure 13.

Baseline with partitions. The original LeanStore paper [26] used a global lock to protect the I/O stage from concurrent access. While a single I/O partition might be good enough for mostly in-memory workloads, in out-of-memory settings this immediately became the bottleneck. To circumvent this, the global lock was removed and exchanged with partitioned page eviction and I/O management, as described in Section 3.5. We use this setting with partitions as our baseline, like in Figure 1. The bottleneck now is the SSD bandwidth. This can be seen in the I/O plots of Figure 13, in both benchmarks the throughput increases to the maximum of what the SSDs can do with 16 KB pages. In terms of IOPS, however, the 31 GB/s in TPC-C and 53 GB/s in read-only correspond to only ≈1.9 M and 3.2 M IOPS, out of 6 M to 12 M the SSD can do with 4 KB pages. Note, that the throughput in GB/s is slightly higher with 16 KB pages than the maximum of what the SSDs can do at 4 KB pages, as shown in the background section.

Page size. To use the full SSD bandwidth in terms of IOPS it is necessary to switch to a smaller page size. Hence, in the next step we are benchmarking LeanStore with +4 KB pages. As expected, this has a big impact on TPC-C performance, which doubles to over 500k tps. This large increase is because TPC-C has a high write percentage (≈40% of total I/O) with which the SSD bandwidth is a lot lower compared to read-only (i.e., ≈26 GB/s). Scaling down the page size by 4× effectively quadruples the throughput in IOPS. The read-only performance increase is more modest. IOPS are far higher than in TPC-C even with 16 KB pages, as the bottleneck is somewhere else (Linux RAID).

CPU optimization and RAID. The discussed +CPU optimizations increase TPC-C performance by over 30%. This directly corresponds to the CPU time saved by the discussed micro-optimization. Similarly, implementing a custom, more efficient +RAID 0 results in a similar performance boost. In the read-only workload the performance improvement is even higher, as the throughput numbers are much higher than in TPC-C and the Linux md raid seems to have a hard limit at around 15 GB/s. Removing this bottleneck has an almost 2× improvement in read-only lookup speed.

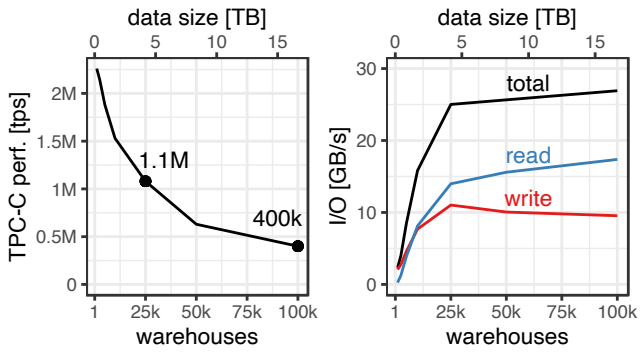


Figure 14: LeanStore out-of-memory scalability (400 GB buffer pool, 64 threads, I/O backend: SPDK).

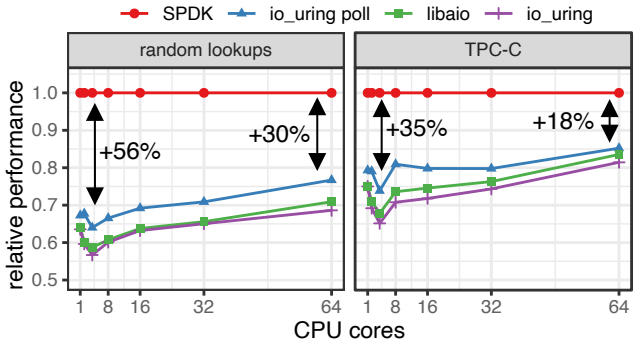


Figure 15: Performance of different I/O backends and limited number of CPU cores (16 GB buffer pool, 160 GB data).

Avoiding over-subscription. All previous experiments were run with high thread over-subscription. The number of background threads and workers was tuned for best performance. This was up to 1500 worker threads and 40 isolated background threads for page eviction. With user-space +tasks this over-subscription and tuning is unnecessary. The removal of this context switching overhead increases TPC-C performance by 16% and read-only by 25%.

User-space I/O. Lastly, we use kernel-bypassing for I/O using +SPDK. In TPC-C this does not increase the performance by a lot, as even with libaio LeanStore is approaching the maximum SSD bandwidth (for mixed read/write workloads). Further, we used the optimum number of threads for every setting, in case for +tasks (1.03 M tps and 10.08 M lookups/s) this was 128 threads. With +spdk (1.07 M tps) it is 64 threads, as that is enough to achieve the highest performance and fully saturate all SSDs. In the random/read-only lookup, we are approaching a bottleneck that interrupt based I/O seems to have on our system, limiting the throughput to about ≈10 M IOPS. With SPDK (13.26 M lookups/s, note: 10% of lookups are buffered) there is no such limit and the full SSD bandwidth can be used. In the TPC-C workload io_uring in I/O poll can achieve the same throughput as SPDK. However, it requires more than 64 threads (1.07 M tps with 128 threads). io_uring achieved slightly less, 12.2 M random lookups/s with 128 threads, showing that the additional CPU cycles required for it are starting to matter at this IOPS/CPU speed ratio.

No high pole in the tent. As can be seen, all optimizations and techniques are necessary to exploit the full potential that fast NVMe arrays offer.

4.4 Out-Of-Memory Scalability

Out-of-memory overhead. LeanStore's in-memory performance in comparable to main-memory database systems. For in-memory workloads, it can achieve over 3 million TPC-C transactions per second (128 threads). Going out-of-memory triggers a lot of additional code being called, including eviction strategy, page faults, the whole I/O path, context switching. Naturally, performance is going to decrease from this additional work, when CPU cycles are limited.

One million. Figure 14 shows the out-of-memory scalability of LeanStore, by increasing the data set size (warehouses) with a fixed size buffer pool of 400 GB. When data fits completely in memory, LeanStore achieves over 2 million tps with 64 threads. Going out-of-memory, the performance gracefully degrades. In a 10× out-of-memory setting with 25k warehouses or 4 TB of data, LeanStore can still get to 1.1 million tps. The full I/O bandwidth (25 GB/s) provided by the 8×NVMe drives in a 56%/44% read/write setting is used in this configuration. To demonstrate LeanStore's ability to scale far beyond that, we tested it with very large database size, up to 100,000 warehouses, which corresponds to 16 TB of data, or an out-of-memory factor of 40×. In this setting, LeanStore still achieves 400 thousand transactions per second.

Running even larger databases would technically be possible as well. However, running the system at 40× out-of-memory is already quite extreme. The price difference per capacity of DRAM to flash SSDs is currently about 20×. Hence, with even larger databases it makes sense to also budget more money for additional DRAM.

4.5 I/O Interfaces

A realistic workload. In this experiment we evaluate the different I/O backends supported by LeanStore. Obviously the choice for the right I/O interface depends on given constraints, e.g., is a file system required, buffered or non-buffered I/O, can the whole device be used exclusively by the application, etc. Our evaluation is solely based on performance and efficiency, with LeanStore representing a more realistic workload than I/O microbenchmarks.

Larger difference in read-only. In Section 4.3 we saw that with TPC-C the additional step of using user-space I/O did not improve performance substantially. Meanwhile, the difference in read-only lookups was relatively large. This can be attributed to the fact that the mixed read/write performance of SSDs is substantially lower than read only (mixed: 6.7M IOPS vs. read-only: 12.5M IOPS, flash writes are around 10× more expensive than reads). Hence, with enough CPU cores available for the workload it is possible to achieve saturation with every interface.

Limiting CPU cores. To show the actual CPU overhead of kernel I/O libraries we compare them to SPDK and limit the number of

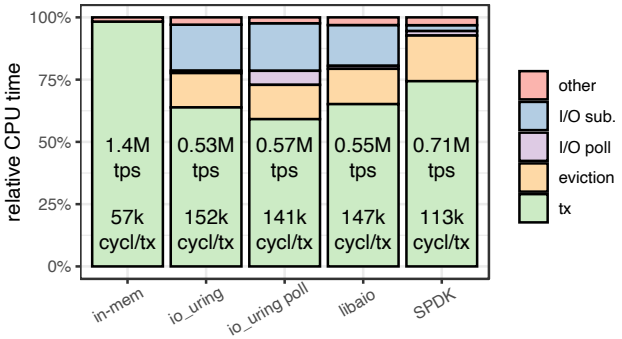


Figure 16: Relative CPU time on database system components with TPC-C transactions (160 GB data, first bar in-memory, others with 16 GB buffer pool, 32 threads).

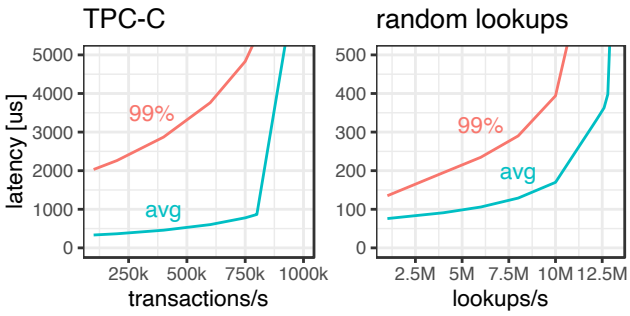


Figure 17: Latency with average and 99th percentile (160 GB database, 16 GB buffer pool, 63 threads, using SPDK).

CPU cores in use. By doing this, we can observe the differential impact on system performance when running on a restricted number of cores, thus revealing the efficiency of different I/O libraries. Figure 15 shows the relative performance in read-only and TPC-C using io_uring, libaio and SPDK, which always achieves the highest performance. The general trend is with fewer CPU cores available the better SPDK performs, as it is more efficient in CPU usage. For the read-only workload SPDK is on average 60%-80% faster than the kernel libraries. With more CPU cores available the difference decreases to around 50%. In TPC-C the differences are smaller as the workload is less I/O heavy compared to read-only lookups. CPU cycles used on I/O are fewer and hence the advantage of using SPDK is smaller. io_uring in default settings and with I/O polling disabled is surprisingly slightly slower (≈2% on average) than libaio. With I/O polling enable io_uring is (≈5-8%) faster than libaio.

Useable flags. io_uring offers a plethora of flags to change and improve its behavior for specific use cases. In Section 2.4 we already mentioned the `IOPOLL` flag that we have been using in the experiments. It should also be noted, that to use polled I/O the kernel nvme module has to be instructed to allocate a number of queues for polling by setting `nvme.poll_queue`. Another flag that was already mentioned is `SQPOLL`. It eliminates the necessity for system calls, as it creates kernel workers that poll the submission queue, handle SSD I/O-submission and polling. We experimented with the setting (including `ATTACH_WQ`, `SQ_AFF`), but it was not possible to get better efficiency in terms of cycles per I/O operation. Further, this flag again introduces hard to manage background threads that we do not want in our design.

Fighting the kernel. At this point it is also important to note that submit batching and reducing poll calls mostly benefit kernel libraries. This is partially because it reduces system calls, but mostly because the overhead to go through the kernel I/O stack (i.e., the block layer) is high. This is currently the same for all kernel I/O interfaces. With SPDK, batching will lower the number of MMIO calls on the NVMe submission doorbell. However, SPDK shows that submission and polling are actually already relatively cheap. Saturating all SSDs in a microbenchmark with SPDK requires only three CPU cores. A SPDK polling call only takes about 80ns (200 cycles). Optimizing against specific kernel characteristics is not a promising approach, especially because there is a lot of active development going on in the Linux kernel I/O subsystem with io_uring and for example, `nvme_passthrough`.

4.6 CPU Usage

The cost of not using SPDK in terms of CPU cycles is relatively high. Figure 16 shows that in in-memory workloads all CPU time is used for actual transactional workload. Going out of memory in the usual setting with 1000 TPC-C warehouses and 16 GB of RAM, a lot of time is spent on eviction and I/O submission. Eviction is very similar in all out-of memory workloads around 14%, except with SPDK it is slightly higher (18%), as it is proportional to the transactional share, which is higher when using SPDK. This is mainly the case because submitting IO requests is much more efficient in SPDK (≈2% vs. 16-19%). This is less a matter of the interface, but the inefficient implementation of the I/O path in the Linux kernel (v5.19.0-26). We run the experiment also with I/O submission batching (+3% in throughput for kernel libraries) and with kernel mitigations disabled (+4% with kernel parameter `mitigations=off`). However, these two settings did not change the relative differences between kernel I/O libraries.

4.7 Latency

We also measured transaction latency, employing a target transaction rate and exponentially distributed inter-arrival times. The results are depicted in Figure 17. In the random lookup workload, the latency corresponds almost entirely to I/O latency. In the TPC-C workload, latency is much higher because each transaction requires on average 3.5 synchronous read operations. TPC-C also includes the asynchronous eviction of dirty pages to storage, with an average of 2.8 page writes per transaction. This contributes to higher tail latency for reads, as they could be stalled by write operations, which take significantly longer on flash.

5 RELATED WORK

We are not aware of any OLTP system that can fully exploit the capabilities of NVMe arrays. Given the low cost and high performance of flash, there is a surprising lack of research on the topic –

in particular in comparison with the amount of research on systems optimized for main memory or persistent memory.

Initial flash bubble. Flash SSDs started to become relevant around 2008, leading to research on how to integrate them into database systems. Research initially focused on whether flash is beneficial for database applications [10, 24, 25]. At that time, performance and price were still orders of magnitude away from where they are now. Hence, flash was primarily used as a cost-efficient extension to the main memory buffer pools [5, 17–19], or for caches of specific database service like recovery [3], multiversion storage [34], update caching [35], or sketches [12]. Given current prices, flash has become the primary persistent storage medium rather than an additional layer.

Specialized hardware. NAND flash has very different physical hardware characteristics than magnetic disks, e.g., flash does not support in-place writes. For backwards-comparability, commercial SSDs emulate disks using complex SSD-internal logic. This can lead to lower-than-necessary and unpredictable performance. A separate line of research, therefore, investigates specialized SSDs [14, 20, 37], novel interfaces [4, 29, 32, 33], and co-designing databases and storage hardware [9, 28] Some systems are optimized to reduce write amplification [11], and several systems [6, 7, 16, 21, 23] have been designed for Optane (3D Xpoint) based memory, which has different hardware characteristics compared to flash. This work, in contrast, is applicable to all cheap, off-the-shelf NVMe SSDs.

High-performance storage engines. There are a number of storage engines and key-value stores optimized for flash. RocksDB [36] is based on an LSM-Tree that is optimized for low write amplification (at the cost of higher read amplification). RocksDB was designed for flash storage, but at the time of SATA SSDs, and therefore cannot saturate large NVMe arrays. The techniques we propose in this paper are general and can be integrated into LSM-based storage engines. Another efficient storage engine is WiredTiger [30], which like our LeanStore system is based on B-trees. One difference between LeanStore and WiredTiger is that the latter uses different representations for in-memory and out-of-memory nodes. This can save space, but the conversion between the representations ("reconciliation") becomes CPU expensive with fast storage devices.

High-performance key value stores. Like our approach, PA-Tree [41] uses SPDK as an efficient asynchronous I/O library, but it relies on a single thread and therefore cannot exploit the full potential of NVMe arrays. Tucana [31] is a key value store based on a B^{ϵ}-Tree optimized for SSD storage. However, it was built on the premise of slow SATA SSDs and relies on `mmap` for I/O, which was already deemed inadequate for NVMe-era performance [8]. KVell [27] comes closest to exploiting full NVMe array bandwidths. Unlike LeanStore, KVell is a partitioned key value store, that does not store keys sorted on disk. This is a big limitation for range queries and small payloads and makes it unfit to be used as a general purpose storage engine for DBMS. Further, even with these limitations, KVell is not able to fully saturate large flash NVMe arrays in their experiments. Indeed, they report being CPU bound and achieving only 75% of the I/O capabilities on a server with 3.3 M IOPS. We therefore believe that to the best of our knowledge, LeanStore with the techniques discussed in this paper is the only system that is currently capable of exploiting the full potential of NVMe arrays.

6 CONCLUSIONS

In this paper, we showed what modern NVMe storage arrays can provide and how it is possible for high-performance storage engines to fully exploit this performance.

We can give answers to the questions posed in the introduction:

- Q1: Arrays of NVMe SSDs can achieve the performance promised in hardware specifications. In fact, we achieved slightly higher throughput at 12.5 M IOPS with our 8×SSD setup.
- Q2: Good performance can be achieved with all asynchronous I/O interfaces. Kernel-bypassing is not essential to achieve full bandwidth even with small pages. However, it is more efficient in CPU usage.
- Q3: The best trade-off between random IOPS, throughput, latency, and I/O amplification is achieved with 4 KB pages.
- Q4: To manage the high parallelism required for large NVMe SSD arrays, the database system must employ a low overhead mechanism to quickly jump between user queries. To solve that we employed cooperative multitasking using lightweight user-space threads.
- Q5: Managing workloads with tens of million IOPS makes out-of-memory code paths hot and performance critical. This requires scalable I/O management through partitioning relevant data structures to prevent contention hotspots. The replacement algorithm has to be optimized to evict tens of millions of pages per second
- Q6: I/O should be performed directly by worker threads. In our design worker threads in fact perform all duties, like, in-memory work, eviction, and I/O.This symmetric design has conceptual advantages and allows for a more robust system.

Hardware and software progress. The Linux kernel I/O interfaces performed surprisingly well and were able to achieve 10M IOPS in our benchmarks. io_uring with I/O polling enabled was the only kernel interface that could achieve the full bandwidth of our NVMe array. The big advantage of kernel-bypassing with SPDK is its efficiency in terms of CPU cycles. However, SPDK also has big disadvantages, for example, it requires root privileges and exclusive access to the whole drive. For now, there is enough CPU time available for the less efficient kernel interfaces. With upcoming PCIe 5.0 SSDs this ratio between CPU cycles and I/O operations is changing again, which might make kernel bypassing more relevant. But kernel development is not standing still and could keep up with this development.

Flash is King. As we have discussed in Section 1, the economic landscape is clear: NVMe SSDs are attractive for high-performance OLTP and will become even more relevant with further increasing bandwidths. As more systems adapt to this economic reality, we believe our work can serve as an important foundation.

ACKNOWLEDGMENTS

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 447457559.

REFERENCES

[1] Adnan Alhomssi, Michael Haubenschild, and Viktor Leis. 2023. The Evolution of LeanStore. In *BTW (LNI)*, Vol. P-331. 259–281.

[2] Jens Axboe. 2023. *fio - Flexible I/O tester.* Retrieved March 1, 2023 from https://github.com/axboe/fio

[3] Bishwaranjan Bhattacharjee, Kenneth A. Ross, Christian A. Lang, George A. Mihaila, and Mohammad Banikazemi. 2011. Enhancing recovery using an SSD buffer pool extension. In *DaMoN*. 10–16.

[4] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. In *FAST*. 359–374.

[5] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. 2010. SSD Bufferpool Extensions for Database Systems. *Proc. VLDB Endow.* 3, 2 (2010), 1435–1446.

[6] Jiajia Chu, Yunshan Tu, Yao Zhang, and Chuliang Weng. 2020. Latte: A Native Table Engine On Nvme Storage. In *ICDE*. 1225–1236.

[7] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard P. Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *USENIX Annual Technical Conference*. 49–63.

[8] Andrew Crotty, Viktor Leis, and Andrew Pavlo. 2022. Are You Sure You Want to Use MMAP in Your Database Management System?. In *CIDR*. www.cidrdb.org.

[9] Jaeyoung Do, Ivan Luiz Picoli, David B. Lomet, and Philippe Bonnet. 2021. Better database cost/performance via batched I/O on programmable SSD. *VLDB J.* 30, 3 (2021), 403–424.

[10] Ming Du, Yan Zhao, and Jiajin Le. 2009. Using Flash Memory as Storage for Read-Intensive Database. In *DBTA*. 472–475.

[11] Athanasios Fevgas, Leonidas Akritidis, Panayiotis Bozanis, and Yannis Manolopoulos. 2020. Indexing in flash storage devices: a survey on challenges, current approaches, and future trends. *VLDB J.* 29, 1 (2020), 273–311.

[12] Mayank Goswami, Dzejla Medjedovic, Emina Mekic, and Prashant Pandey. 2018. Buffered Count-Min Sketch on SSD: Theory and Experiments. *CoRR* abs/1804.10673 (2018).

[13] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*.

[14] Sergey Hardock, Ilia Petrov, Robert Gottstein, and Alejandro P. Buchmann. 2015. NoFTL for Real: Databases on Real Native Flash Storage. In *EDBT*. 517–520.

[15] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *SIGMOD Conference*. ACM, 877–892.

[16] Kaisong Huang, Darien Imai, Tianzheng Wang, and Dong Xie. 2022. SSDs Striking Back: The Storage Jungle and Its Implications to Persistent Indexes. In *CIDR*.

[17] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. 2012. Flash-based Extended Cache for Higher Throughput and Faster Recovery. *Proc. VLDB Endow.* 5, 11 (2012), 1615–1626.

[18] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. 2016. Flash as cache extension for online transactional workloads. *VLDB J.* 25, 5 (2016), 673–694.

[19] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. 2014. Durable write cache in flash memory SSD for relational and NoSQL databases. In *SIGMOD Conference*. 529–540.

[20] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. 2014. Supporting Transactional Atomicity in Flash Storage Devices. *IEEE Data Eng. Bull.* 37, 2 (2014), 27–34.

[21] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the performance of fast NVM storage with uDepot. In *FAST*. 1–15.

[22] Oliver Kowalke. 2014. *Boost Context Library.* Retrieved March 1, 2023 from https://www.boost.org/doc/libs/1_80_0/libs/context/doc/html/index.html

[23] Sangjin Lee, Alberto Lerner, André Ryser, Kibin Park, Chanyoung Jeon, Jinsub Park, Yong Ho Song, and Philippe Cudré-Mauroux. 2022. X-SSD: A Storage System with Native Support for Database Logging and Replication. In *SIGMOD Conference*. 988–1002.

[24] Sang-Won Lee, Bongki Moon, and Chanik Park. 2009. Advances in flash memory SSD technology for enterprise database applications. In *SIGMOD Conference*. 863–870.

[25] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. 2008. A case for flash memory ssd in enterprise database applications. In *SIGMOD Conference*. 1075–1086.

[26] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. 185–196.

[27] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key-value store. In *SOSP*. 447–461.

[28] Alberto Lerner and Philippe Bonnet. 2021. Not your Grandpa's SSD: The Era of Co-Designed Storage Devices. In *SIGMOD Conference*. 2852–2858.

[29] Leonardo Mármol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *USENIX Annual Technical Conference*. 207–219.

[30] MongoDB. 2022. *WiredTiger - WiredTiger is an high performance, scalable, production quality, NoSQL, Open Source extensible platform for Data management.* Retrieved March 1, 2023 from https://source.wiredtiger.com/

[31] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2016. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *USENIX Annual Technical Conference*. 537–550.

[32] Ivan Luiz Picoli, Niclas Hedam, Philippe Bonnet, and Pinar Tözün. 2020. Open-Channel SSD (What is it Good For). In *CIDR*.

[33] Devashish R. Purandare, Peter Wilcox, Heiner Litz, and Shel Finkelstein. 2022. Append is Near: Log-based Data Management on ZNS SSDs. In *CIDR*.

[34] Mohammad Sadoghi, Kenneth A. Ross, Mustafa Canim, and Bishwaranjan Bhattacharjee. 2013. Making Updates Disk-I/O Friendly Using SSDs. *Proc. VLDB Endow.* 6, 11 (2013), 997–1008.

[35] Mohammad Sadoghi, Kenneth A. Ross, Mustafa Canim, and Bishwaranjan Bhattacharjee. 2016. Exploiting SSDs in operational multiversion databases. *VLDB J.* 25, 5 (2016), 651–672.

[36] Meta Open Source. 2022. *RocksDB - A persistent key-value store for fast storage environments.* Retrieved March 1, 2023 from http://rocksdb.org/

[37] Yanjie Tan, Huailiang Tan, Peng Zhu, Youyou Lu, and Zaihong He. 2022. Embedded Transaction Support Inside SSD With Small-Capacity Non-Volatile Disk Cache. *IEEE Trans. Knowl. Data Eng.* 34, 5 (2022), 2148–2163.

[38] Linux Kernel Team. 2019. *Efficient IO with io_uring.* Retrieved March 1, 2023 from https://kernel.dk/io_uring.pdf

[39] SPDK Team. 2023. *Storage Performance Development Kit (SPDK).* Retrieved March 1, 2023 from https://spdk.io/

[40] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Building blocks for persistent memory. *VLDB J.* 29, 6 (2020), 1223–1241.

[41] Li Wang, Zining Zhang, Bingsheng He, and Zhenjie Zhang. 2020. PA-Tree: Polled-Mode Asynchronous B+ Tree for NVMe. In *ICDE*. 553–564.