Virtual-Memory Assisted Buffer Management

Preprint accepted for publication at SIGMOD 2023

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

Adnan Alhomssi
Friedrich-Alexander-Universität Erlangen-Nürnberg
adnan.alhomssi@fau.de

Tobias Ziegler
Technische Universität Darmstadt
tobias.ziegler@cs.tu-darmstadt.de

Yannick Loeck
Technische Universität Hamburg
yannick.loeck@tuhh.de

Christian Dietrich
Technische Universität Hamburg
christian.dietrich@tuhh.de

ABSTRACT

Most database management systems cache pages from storage in a main memory buffer pool. To do this, they either rely on a hash table that translates page identifiers into pointers, or on pointer swizzling which avoids this translation. In this work, we propose *vmcache*, a buffer manager design that instead uses hardware-supported virtual memory to translate page identifiers to virtual memory addresses. In contrast to existing `mmap`-based approaches, the DBMS retains control over page faulting and eviction. Our design is portable across modern operating systems, supports arbitrary graph data, enables variable-sized pages, and is easy to implement. One downside of relying on virtual memory is that with fast storage devices the existing operating system primitives for manipulating the page table can become a performance bottleneck. As a second contribution, we therefore propose *exmap*, which implements scalable page table manipulation on Linux. Together, vmcache and exmap provide flexible, efficient, and scalable buffer management on multi-core CPUs and fast storage devices.

CCS CONCEPTS

• **Information systems** → **Data management systems**; **Record and buffer management**.

KEYWORDS

Database Management Systems; Operating Systems; Caching; Buffer Management

ACM Reference Format:
Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management: Preprint accepted for publication at SIGMOD 2023. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

DBMS vs. OS. Database management systems (DBMS) and operating systems (OS) have always had an uneasy relationship. OSs provide process isolation by virtualizing hardware access, whereas DBMSs want full control over hardware for optimal efficiency. At the same time, OSs offer services (e.g., caching pages from storage) that are *almost* exactly what database systems require – but for performance and semantic reasons, DBMSs often re-implement this functionality. The mismatch between the services offered by operating systems and the requirements of database systems was raised four decades ago [40], and the situation has not improved much since then.

OS-controlled caching. The big advantage the OS has over a DBMS is that it runs in kernel mode and therefore has access to privileged instructions. In particular, the OS has direct control over the virtual memory page table, and can therefore do things user space processes cannot. For example, using virtual memory and the memory management unit (MMU) of the processor, the OS implements transparent page caching and exposes this by mapping storage into virtual memory through the `mmap` system call. With `mmap`, in-memory operations (cache hits) are fast, thanks to the Translation Lookaside Buffer (TLB). Nevertheless, as Crotty et al. [13] recently discussed, `mmap` is generally *not* a good fit for database systems. Two major problems of `mmap` are that (1) the DBMS loses control over page faulting and eviction, and that (2) the virtual memory implementation in Linux is too slow for modern NVMe SSDs [13]. The properties of `mmap` and alternative buffer manager designs are summarized in Table 1.

DBMS-controlled caching. In order to have full control, most DBMSs therefore avoid file-backed `mmap`, and implement explicit buffer management in user space. Traditionally, this has been done using a hash table that contains all pages that are currently in cache [15]. Recent, more efficient buffer manager designs rely on pointer swizzling [16, 23, 33]. Both approaches have downsides: the former has non-trivial hash table translation overhead; and the latter is more difficult to implement and does not support cyclical page references (e.g., graph data). Rather than compromising on either the performance or the functionality benefits of translation, this work proposes hardware-supported virtual memory as a fundamental building block of buffer management.

Contribution 1: vmcache. The first contribution of this paper is *vmcache*, a novel buffer pool design that relies on virtual memory, but retains control over faulting and eviction within the DBMS,

Table 1: Conceptual comparison of buffer manager designs

	mmap [13]	tradi. [15]	pointer swiz. [16, 23]	Umbra [33]	*vmcache* Sec. 3	*+exmap* Sec. 4
transl.	page tbl.	hash tbl.	invasive	invasive	page tbl.	page tbl.
control	OS	DBMS	DBMS	DBMS	DBMS	DBMS
var. size	easy	hard	hard	med. (*)	easy	easy
graphs	yes	yes	no	no	yes	yes
implem.	med. (**)	easy	hard	hard	easy	easy
in-mem.	fast	slow	fast	fast	fast	fast
out-mem.	slow	fast	fast	fast	med.	fast

(*) only powers of 2 [33] (**) read-only easy, transactions hard [13]

unlike solutions based on file-backed mmap. The key idea is to map the storage device into anonymous (rather than file-backed) virtual memory and use the MADV_DONTNEED hint to explicitly control eviction. This enables fast in-memory page accesses through TLB-supported translations without handing control to the OS. Page-table-based translation also allows vmcache to support arbitrary graph data and variable-sized pages.

Contribution 2: exmap. While vmcache has excellent in-memory performance, every page fault and eviction involves manipulating the page table. Unfortunately, existing OS page table manipulation primitives have scalability problems that become visible with high-performance NVMe SSDs [13]. Therefore, as a second contribution, we propose *exmap*, an OS extension for efficiently manipulating virtual memory mappings. exmap is implemented as a Linux kernel module and is an example of DBMS/OS co-design. By providing new OS-level abstractions, we simplify and accelerate data-processing systems. Overall, as Table 1 shows, combining exmap with vmcache results in a design that is not only fast (in-memory and out-of-memory) but also offers important functionality.

2 BACKGROUND: DATABASE PAGE CACHING

Buffer management. Most DBMSs cache fixed-size pages (usually 4-64 KB) from secondary storage in a main memory pool. The basic problem of such a cache is to efficiently translate a page identifier (PID), which uniquely determines the physical location of each page on secondary storage, into a pointer to the cached data content. In the following, we describe known ways of doing that, including the six designs shown in Table 1.

Hash table-based translation. Figure 1a illustrates the traditional way [15] of implementing a buffer pool: a hash table indexes all cached pages by their PID. A page is addressed using its PID, which always involves a hash-table lookup. On a miss, the page is read from secondary storage and added to the hash table. This approach is simple and flexible. The hash table is the single source of truth of the caching state, and pages can reference each other arbitrarily through PIDs. The downside is suboptimal in-memory performance, as even cache hits have to pay the hash table lookup cost. Also note that there are two levels of translation: from PID to virtual memory pointer (at the DBMS level), and from virtual memory pointer to physical memory pointer (at the OS/MMU level).

Main-memory DBMS. One way to avoid the overhead of traditional buffer managers is to forego caching altogether and keep all data in main memory. While pure in-memory database systems can be very fast, in the past decade DRAM prices have almost stopped decreasing [18]. Storage in the form of NVMe flash SSDs, on the other hand, has become cheap ($20-50\times$ cheaper per byte than DRAM [18]) and fast (>1 million random 4 KB reads per second per SSD [4]). This makes pure in-memory systems economically unattractive [29], and implies that modern storage engines should combine DRAM and SSD. The challenge is supporting very large data sets on NVMe SSDs with their high I/O throughput *and* making cache hits almost as fast as in main-memory systems.

Pointer swizzling (invasive translation). An efficient technique for implementing buffer managers is pointer swizzling. The technique has originally been proposed for object-oriented DBMSs [20], but has recently been applied to several high-performance storage engines [16, 23, 33]. As Figure 1b illustrates, the idea is to replace the PID of a cached page with its virtual memory pointer *within the data structure*. Page hits can therefore directly dereference a pointer instead of having to translate it through a hash table first. One way to think about this is that pointer swizzling gets rid of explicit hash table-based translation by invasively modifying the data structure itself. Pointer swizzling offers very good in-memory performance. However, it requires adaptations for every buffer-managed data structure, and its internal synchronization is quite intricate. E.g., to unswizzle a page, one needs to find and lock its parent, and Storing a parent pointer on each node presents synchronization challenges during node splits. Another downside is that pointer swizzling-based systems generally do not support having more than one incoming reference to any particular page. In other words, only tree data structures are directly supported. Graph data, next pointers in B+tree leaf pages, and multiple incoming tuple references (e.g., from secondary indexes) require inelegant and sometimes inefficient workarounds.

Hardware-supported page translation. Traditional buffer managers and pointer swizzling present an unsatisfactory and seemingly inescapable tradeoff: either one pays the performance cost of the hash table indirection, or one loses the ability to support graph-like data. Instead of getting rid of the translation (as pointer swizzling does), another way of achieving efficiency is to make PID-to-pointer translation efficient through hardware support. All modern operating systems use virtual memory and, together with hardware support from the CPU, transparently translate virtual to physical addresses. Page table entries are cached within the CPU, in particular the TLB, which makes virtual memory translation fast. Figure 1c shows how hardware-supported page translation can be used for caching pages from secondary storage.

OS-driven caching with file-backed mmap. Unix offers the mmap system call to access storage via virtual memory. After mapping a file or device into virtual memory, a memory access will trigger a page fault. The OS will then install that page in the page table, making succeeding page accesses as fast as ordinary memory accesses. Some systems therefore eschew implementing a buffer pool and instead rely on the OS page cache by mmaping the database file/device. While this approach makes cache hits very fast, it has major problems that were recently analyzed by Crotty et al. [13]: (1) Ensuring transactional safety is difficult and potentially inefficient because the DBMS loses control over eviction. (2) There is no interface for asynchronous I/O, and I/O stalls are unpredictable. (3) I/O error handling is cumbersome. (4) OS-implemented page faulting and

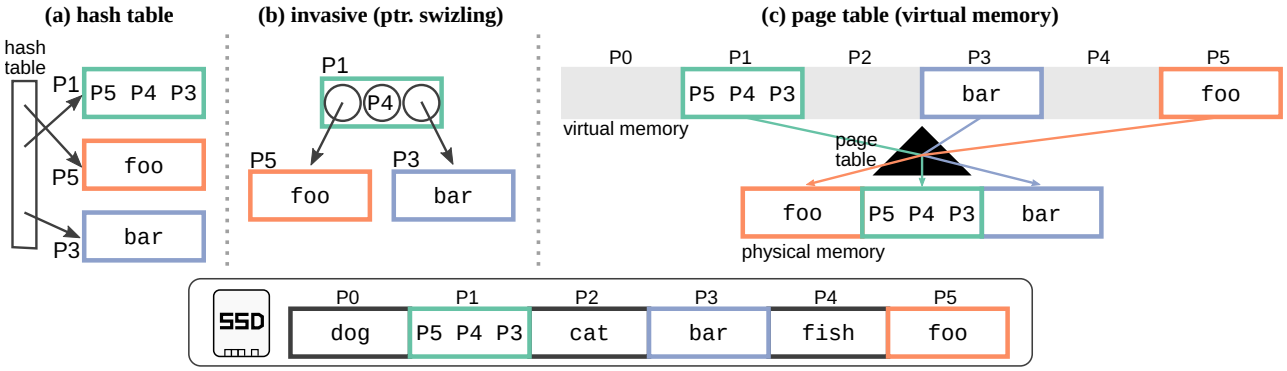


Figure 1: Buffer pool page translation schemes. Example with 6 pages on storage (P0-P5), 3 of which are cached (P1, P3, P5)

eviction is too slow to fully exploit modern NVMe storage devices. The lack of control over eviction for file-backed `mmap` approaches is a fundamental problem. Notably, it prevents the implementation of ARIES-style transactions. ARIES uses in-place writes and prevents the eviction of a dirty page before its corresponding log entry is flushed – impossible with existing OS interfaces [13]. Without explicit control over eviction, it is also impossible to implement DBMS-optimized page replacement algorithms. Thus, one is at the whim of whatever algorithm the OS currently in use implements, which is unlikely to be optimized for DBMS workloads.

DBMS-driven, virtual-memory assisted caching. While OS-managed caching using `mmap` may not be a good solution for most DBMSs, the OS has one big advantage: instead of having to use an explicit hash table for page translation, it can rely on hardware support (the TLB) for page translation. This raises the following question: Is it possible to exploit the virtual memory subsystem without losing control over eviction and page fault handling? One contribution of this paper is to answer this question affirmatively. In Section 3, we describe how widely-supported OS features (anonymous memory and the `MADV_DONTNEED` hint) can be exploited to implement hardware-supported page translation while retaining full control over faulting and eviction within the DBMS.

Variable-sized pages. Besides making page translation fast, using a page table also makes implementing multiple page sizes much easier. Having dynamic page sizes is obviously very useful, e.g., for storing objects that are larger than one page [33]. Nevertheless, many buffer managers only support one particular page size (e.g., 4 KB) because multiple sizes lead to complex allocation and fragmentation issues. In these systems, larger objects need to be implemented by splitting them across pages, which complicates and slows down the code accessing such objects. With control over the page table, on the other hand, a larger (e.g., 12 KB) page can be created by mapping multiple (e.g., 3) non-contiguous physical pages to a contiguous virtual memory range. This is easy to implement within the OS and no fragmentation occurs in main memory. One system that allows multiple (albeit only power-of-two) page sizes is Umbra [33]. It implements this by allocating multiple buffer pool-sized virtual memory areas – one for each page size. To allocate a page of a particular size, one can simply fault the memory from that class. To free a page, the buffer manager uses the `MADV_DONTNEED` OS hint. This approach gets rid of fragmentation from different page sizes, but Umbra's page translation is still based on pointer swizzling rather than the page table. Umbra therefore inherits the disadvantages of pointer swizzling (difficult implementation, no graph data), while potentially encountering OS scalability issues.

Fast virtual memory manipulation. While OS-supported approaches offer very fast access to cached pages and enable variable-sized pages, they unfortunately may suffer from performance problems. One problem is that each CPU core has its own TLB, which can get out of sync with the page table[1]. When the page table changes, the OS therefore generally has to interrupt all CPU cores and force them to invalidate their TLB ("TLB shootdown"). Another issue is that intra-kernel data structures can become the scalability bottleneck on systems with many cores. Crotty et al. [13] observed that because of these issues `mmap` can be slow in out-of-memory workloads. For random reads from one SSD, they measured that it achieves less than half the achievable I/O throughput. With sequential scans from ten SSDs, the gap between `mmap` and explicit asynchronous I/O is roughly 20×. Any virtual memory-based approach (including our basic vmcache design) will run into these kernel issues. Section 4 therefore describes a novel, specialized virtual memory subsystem for Linux called *exmap*, which solves these performance problems.

Persistent memory. In this work, we focus on block storage rather than byte-addressable persistent memory, for which multiple specialized caching designs have been proposed [8, 21, 28, 41, 43].

3 VMCACHE: VIRTUAL-MEMORY ASSISTED BUFFER MANAGEMENT

The POSIX system call `mmap` usually maps a file or storage device into virtual memory, as is illustrated in Figure 1c. The advantage of file-backed `mmap` is that, due to hardware support for page translation, accessing cached pages becomes as fast as ordinary memory accesses. If the page translation is cached in the TLB and the data happens to be in the L1 cache, an access can take as little as 1 ns. The big downside is that the DBMS loses control over page faulting and eviction. If the page is not cached but resides on storage, dereferencing a pointer may suddenly take 10 ms because the OS

[1]The page table, which is an in-memory data structure, itself is coherent across CPU cores. However, a CPU core accessing memory caches virtual to physical pointer translations in a per-core hardware cache called TLB. If the page table is changed, the hardware does not automatically update or invalidate existing TLB entries.

will cause a page fault that is transparent to the DBMS. Thus, from the point of view of the DBMS, eviction and page faulting are totally unpredictable and can happen at any point in time. In this section, we describe vmcache, a buffer manager design that – like file-backed mmap – uses virtual memory to translate page identifiers into pointers (see Figure 1c). However, unlike mmap, in vmcache the DBMS retains control over page faults and eviction.

3.1 Page Table Manipulation

Setting up virtual memory. Like the file-backed mmap approach, vmcache allocates a virtual memory area with (at least) the same size as the backing storage. However, unlike with file-backed mmap this allocation is not directly backed by storage. Such an "unbacked" allocation is called anonymous and, confusingly, is done through mmap as well, but using the MAP_ANONYMOUS flag:

```
int flags = MAP_ANONYMOUS|MAP_PRIVATE|MAP_NORESERVE;
int prot = PROT_READ | PROT_WRITE;
char* virtMem = mmap(0, vmSize, prot, flags, -1, 0);
```

Note that no file descriptor has been specified here (the fourth argument is -1). Storage is handled explicitly and could be a file (multiple applications share one file system) or multiple block devices (in a RAID setup). Moreover, the allocation will initially not be backed by physical memory, which is important because storage capacity is usually much larger than main memory.

Adding pages to the cache. To add a page to the cache, the buffer manager explicitly reads it from storage to the corresponding position in virtual memory. For example, we can use the pread system call to explicitly read P3 as follows:

```
uint64_t offset = 3 * pageSize;
pread(fd, virtMem + offset, pageSize, offset);
```

Once pread completes, a physical memory page will be installed in the page table and the data becomes visible to the DBMS process. In contrast to mmap, which handles page misses transparently without involving the DBMS, with the vmcache approach the buffer manager controls I/O. For example, we can use either the synchronous pread system call or asynchronous I/O interfaces such as libaio or io_uring.

Removing pages from the cache. After mapping more and more pages, the buffer pool will eventually run out of physical memory, causing failing allocations or swapping. Before that happens, the DBMS needs to start evicting pages, which on Linux can be done as follows[2]:

```
madvise(virtMem + offset, pageSize, MADV_DONTNEED);
```

This call will remove the physical page from the page table and make its physical memory available for future allocations. If the page is dirty (i.e., has been changed), it first needs to be written back to storage, e.g., using pwrite:

```
pwrite(fd, virtMem + offset, pageSize, offset);
```

With the primitives described above, the DBMS can control all buffer management decisions: how to read pages, which pages to evict[3], whether and when to write back a page, and when to remove a page from the page table.

[2]On Windows these primitives are available as VirtualAlloc(..., MEM_RESERVE, ...) and VirtualFree(..., MEM_RELEASE).

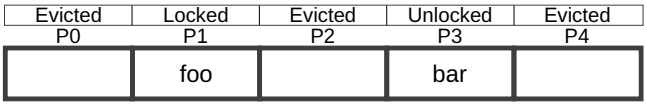
```
fix(uint64_t pid): // fix page exclusively
   uint64_t ofs = pid * pageSize
   while (true) // retry until success
      PageState s = state[pid]
      if (s.isEvicted())
         if (state[pid].CAS(s, Locked))
            pread(fd, virtMem+ofs, pageSize, ofs)
            return virtMem+ofs // page miss
      else if (s.isMarked() || s.isUnlocked())
         if (state[pid].CAS(s, Locked))
            return virtMem+ofs // page hit
unfix(uint64_t pid):
   state[pid].setUnlocked()
```

Listing 1: Pseudo code for exclusive page access

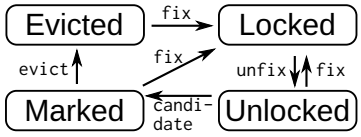
3.2 Page States and Synchronization Basics

In terms of the buffer manager implementation, the most difficult aspect is synchronization, e.g., managing races to the same page. Buffer managers must not only use scalable synchronization internally, they should also provide efficient and scalable synchronization primitives to the upper DBMS layers. After all, most database data structures (e.g., relations, indexes) are stored on top of cacheable pages.

Buffer pool state. In a traditional buffer manager (see Figure 1a), the translation hash table is used as a single source of truth for the caching state. Because all accesses go through the hash table, synchronization is fairly straightforward (but usually not efficient). Our approach, in contrast, needs an additional data structure for synchronization because not all page accesses traverse the page table[4] and because the page table cannot be directly manipulated from user space. Therefore, we allocate a contiguous array with as many page state entries as we have pages on storage at corresponding positions, as the following figure illustrates:



Page states. After startup, all pages are in the Evicted state. Page access operations first check their state entry and proceed according to the following state diagram:



[3]Strictly speaking, the OS could decide to evict vmcache pages – but this does not affect the correctness of our design. OS-triggered eviction can be prevented by disabling swapping or by mlocking the virtual memory range.

[4]If a page translation is cached in the TLB of a particular thread, the thread does not have to consult the page table.

```
1  optimisticRead(uint64_t pid, Function fn):
2    while (true) // retry until success
3      PageState s = state[pid] // incl. version
4      if (s.isUnlocked())
5        // optimistic read:
6        fn(virtMem + (pid*pageSize))
7        if (state[pid] == s) // validate version
8          return // success
9      else if (s.isMarked())
10       // clear mark:
11       state[pid].CAS(s, Unlocked)
12     else if (s.isEvicted())
13       fix(pid); unfix(pid) // handle page miss
```

Listing 2: Pseudo code for optimistic read

Listing 1 shows pseudo code for the `fix` and `unfix` operations, which provide exclusive page access. Suppose we have a page that is currently in `Evicted` state (line 5 in the code). If a thread wants to access that page, it calls `fix`, which will transition to the `Locked` state using a compare-and-swap operation (line 6). The thread is then responsible to read the page from storage and implicitly (via `pread`) install it to the page table (line 7). After that, it can access the page itself and finally `unfix` it, which causes a transition to the `Unlocked` state (line 13). If another thread concurrently wants to fix the same page, it waits until it is unlocked. This serializes page misses and prevents the same page from being read multiple times. The fourth state, `Marked`, helps to implement a clock replacement strategy – though arbitrary other algorithms could be implemented as well. Cached pages are selected for eviction by setting their state to `Marked`. If the page is accessed, it transitions back to the `Locked` state, which clears the mark (line 10). Otherwise, the page can be evicted and eventually transitions to the `Evicted` state.

3.3 Advanced Synchronization

So far, we discussed how to lock pages exclusively. To enable scalable and efficient read operations, vmcache also provides shared locks (multiple concurrent readers on the same page) and optimistic (lock-free) reads.

Shared locks. To implement shared locks for read-only operations, we count the number of concurrent readers within the page state. If the page is not locked exclusively, read-only operations atomically increment/decrement that counter [9] when fixing/unfixing the page. Exclusive accesses have to wait until the counter is 0 before acquiring the lock.

Optimistic reads. Both exclusive and shared locks write to shared memory when acquiring or releasing the lock, which invalidates cache entries in other CPU cores. For tree data structures such as B-trees this results in suboptimal scalability, because the page states of inner nodes are constantly invalidated. An elegant alternative to locks are optimistic, lock-free page reads that validate whether the read was correct. To do that, locks contain an update version that is incremented whenever an exclusively locked page is unlocked [9, 25, 30]. We store this version counter together with the page state within the same 64-bit value, ensuring that both are always changed atomically. As the pseudo code in Listing 2 shows, an optimistic reader retrieves the state and if it equals `Unlocked` (line 4 in the code), it reads from the page (line 5). After that we retrieve the page state again and make sure that the page is still not locked and that the version has not changed (line 6). If this check fails, the operation is restarted. Note that the version counter is incremented not just when a page changes but also when it is evicted. This is crucial for correctness and, for example, ensures that an optimistic read of a marked page that is evicted before validation will fail. To prevent starvation due to repeated restarts, it is also possible to fall back to pessimistic lock-based operations (not shown in the code). Finally, let us note that optimistic reads can be interleaved across multiple pages, enabling lock coupling-like synchronization of complex data structures like B-trees [24]. This approach has been shown to be highly scalable and outperform lock-free data structures [42].

64-bit state entry. Overall, we use 64 bits for the page state, of which 8 bits encode the `Unlocked` (0), `LockedShared` (1-252), `Locked` (253), `Marked` (254), and `Evicted` (255) states. This leaves us with 56 bits for the version counter – which are enough to never overflow in practice. 64 bits are also a convenient size that allows atomic operations such as compare-and-swap (CAS).

Memory reclamation and optimistic reads. In general, lock-free data structures require special care when freeing memory [25, 27, 30]. Techniques such as epoch-based memory reclamation [30] or hazard pointers [31] have been proposed to address this problem. All these techniques incur overhead and may cause additional memory consumption due to unnecessarily long reclamation delays. Interestingly, vmcache – despite supporting optimistic reads – can sidestep these problems completely. Indeed, vmcache does not prevent the eviction/reclamation of a page that is currently read optimistically. However, this is not a problem because after the page is removed from the page table using the `MADV_DONTNEED` hint, it is replaced by the zero page. In that situation the optimistic read will proceed loading 0s from the page without crashing, and will detect that eviction occurred during the version check. (The check fails because eviction first locks and then unlocks the page, which increments the version.) Therefore, vmcache does not need any additional memory reclamation scheme.

Parking lot. To avoid exclusive and shared locks from wasting CPU cycles and ensure fairness under lock contention, one can use the *Parking Lot* [9, 36] technique. The key idea is that if a thread fails to acquire the lock (potentially after trying several times), it can "park" itself, which will block the thread until it is woken up by the thread holding the lock. Parking itself is implemented using a fixed-size hash table storing standard OS-supported condition variables [9]. Within the page state, we only need one additional bit that indicates whether there are threads that are currently waiting for that page lock to be released. The big advantage of parking lots is very low space overhead per page, which is only 1 bit instead of 64 bytes for pthread (rw)locks [9].

3.4 Replacement Strategy

Clock implementation. In principle, arbitrary replacement strategies can be implemented on top of vmcache. As mentioned earlier, our current implementation uses the clock algorithm. Before the buffer pool runs out of memory, we change the state of `Unlocked`

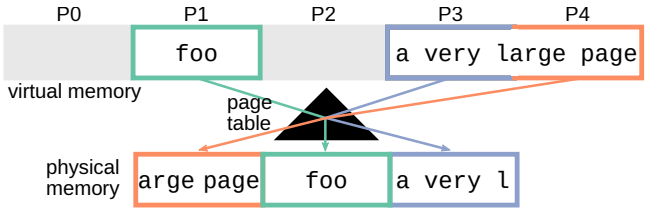


Figure 2: vmcache enables DBMS page sizes that are multiple of the VM page size

pages to `Marked`. All page accesses, including optimistic reads, clear the `Marked` state, ensuring that hot pages will not be evicted. To implement clock, one needs to be able to iterate over all pages in the buffer pool. One approach to do that would be to iterate over the state array while ignoring evicted pages. However, this would be quite expensive if the state array is very sparse (i.e., storage is much larger than main memory). We implement a more robust approach that stores all page identifiers that are currently cached in a hash table. The size of the hash table is equal to the number of pages in DRAM (rather than storage) and our page replacement algorithm iterates over this much smaller data structure. We use a fixed-size open addressing hash table, which makes iteration cache efficient. Note that, in contrast to traditional buffer managers, this hash table is not accessed during cache hits, but only during page faults and eviction.

Batch eviction. For efficiency reasons, our implementation evicts batches of 64 pages. To minimize exclusive locking and exploit efficient bulk-I/O, eviction is done in five steps:

(1) get batch of marked candidates from hash table, lock dirty pages in shared mode
(2) write dirty pages (using `libaio`)
(3) try to lock (upgrade) clean (dirty) page candidates
(4) remove locked pages from page table using `madvise`
(5) remove locked pages from eviction hash table, unlock them

After step 3, all pages must be locked exclusively to avoid race conditions during eviction. For dirty pages, we already obtained shared locks in step 1, which is why step 3 performs a lock upgrade. Clean pages have not been locked, so step 3 tries to acquire the exclusive lock directly. Both operations can fail because another thread accessed the page, in which case eviction skips it (i.e., the page stays in the pool). With the basic vmcache design, step 4 is simply calling `madvise` once for every page. With exmap, we will be able to exploit bulk removal of pages from the page table.

3.5 Page Sizes

Default page size. Most processors use 4 KB virtual memory pages by default, and conveniently this granularity also works well with flash SSDs. It therefore makes sense to set the default buffer pool page size to 4 KB as well. x86 (ARM) also supports 2 MB (1 MB) pages, which might be a viable alternative in systems that primarily read larger blocks. With vmcache, OLTP systems should generally use 4 KB pages and for OLAP systems both 4 KB and 2 MB pages are suitable.

Supporting larger pages. vmcache also makes it easy to support any buffer pool page size that is a multiple of 4 KB. Figure 2 shows an example where page P3 spans two physical pages. For data structures implemented on top of the buffer manager this fact is completely transparent, i.e., the memory appears to be contiguous. Accesses to large pages only use the page state of the head page (P3 not P4 in the figure). The advantage of relying on virtual memory to implement multiple page sizes is that it avoids main memory fragmentation. Note that fragmentation is not simply moved from user to kernel space, but the page table indirection allows the OS to always deal with 4 KB pages rather than having to maintain different allocation classes. As a consequence, as Figure 2 illustrates, a contiguous virtual memory range will in general not be physically contiguous.

Advantages of large pages. Although most DBMS rely on fixed-size pages, supporting different page sizes has many advantages. One case where variable-size pages simplify and accelerate the DBMS is string processing. With variable-size pages one can, for example, simply call external string processing libraries with a pointer into the buffer pool. Without this feature, any string operation (comparison, LIKE, regexp search, etc.) needs to explicitly deal with strings chunked across several pages. Because few existing libraries support chunking, one would have to copy larger strings into a contiguous memory before being able to use them. Another case is compressed columnar storage where each column chunk has the same number of tuples but a different size. In both cases it is indeed possible to split the data across multiple fixed-size pages (and many systems have to do it due to a lack of variable-size support), but it leads to complex code and/or slower performance. Finally, let us mention that, in contrast to systems like Umbra [33], vmcache supports arbitrary page sizes as long as they are a multiple of 4 KB. This reduces memory waste for larger objects. Overall, we argue that this feature can substantially simplify the implementation of the DBMS and lead to better performance.

3.6 Discussion

State access. As mentioned earlier, every page access must retrieve the page state – often causing a cache miss – before it can read the page data itself. One may therefore wonder whether this is just as inefficient as traditional hash table-based buffer managers. However, these two approaches are very different from each other in terms of their memory access patterns. In the hash table approach, the page data pointer is retrieved from the hash table itself, i.e., there is a data dependency between the two pointers and one usually pays the price of two cache miss latencies. In our approach, in contrast, both the page state pointer and data content pointer are known upfront. As a consequence, the out-of-order execution of modern CPUs will perform both accesses in parallel, hiding the additional overhead of the state retrieval.

Memory consumption. vmcache comes with some DRAM overhead in the form of page tables and the page state array: For configuring the virtual-memory mapping, vmcache requires 8.016 bytes for each 4 KB of storage to set up a 5-level page table. Besides this cost, which is inherent to any `mmap`-like buffer manager, vmcache requires additional 8 bytes for the page state: 8 bits for the exclusive/shared lock and 56 bits for the optimistic-read version counter. So in total, vmcache requires around 16 bytes of DRAM per 4 KB on storage. Thus, for example, for 1 TB flash SSD, one needs 4 GB of

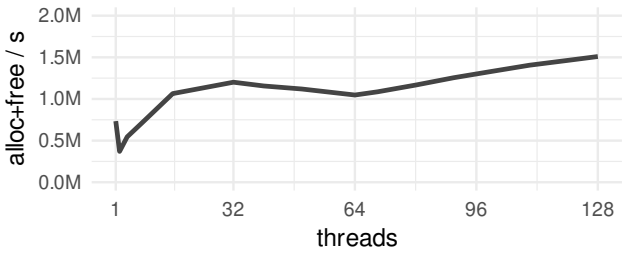


Figure 3: Linux page (de)allocation performance

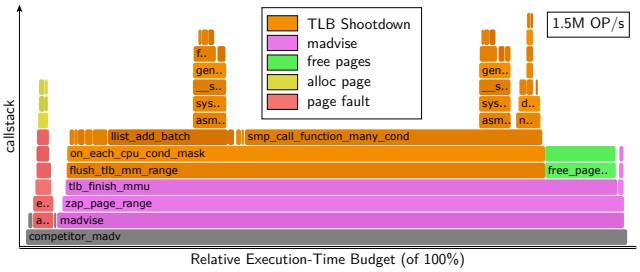


Figure 4: CPU time profile for Figure 3 with 128 threads

DRAM for the internal buffer manager state, which is a reasonable $\frac{1}{256}$th of SSD capacity. Economically speaking, as Flash is approximately 50× cheaper per byte than DRAM, the additional memory costs $\frac{50}{256} \approx 20\,\%$ of the flash price. While this is low enough in most use cases, there are ways to reduce this cost: (1) Compress the 64-bit page state at the expense of optimistic reads (-56 bits) and shared locking (-6 bits) down to two bits per storage page (evicted, exclusive locked), leaving us with a total of 2.07 GB for a 1 TB flash SSD (+10.11 % cost). (2) Place the page state within the buffered page and keep the corresponding 8 bytes on the storage page unused, leaving us with the unavoidable 2 GB of DRAM overhead. Thus, the memory overhead is reasonable in terms of overall cost for the system and could be reduced even further.

Address space. Existing 64-bit CPUs generally support at least 48-bit virtual memory addresses. On Linux, half of that is reserved for the kernel, and user-space virtual memory allocations are therefore limited to $2^{47} = 128$ TB. Starting with Ice Lake, Intel processors support 57-bit virtual memory addresses, enabling a user-space address space size of $2^{56} = 64$ PB. Thus, the address space is large enough for our approach, and will be so for the foreseeable future.

4 EXMAP: SCALABLE AND EFFICIENT VIRTUAL MEMORY MANIPULATION

vmcache exploits hardware-supported virtual memory with explicit control over eviction while supporting flexible locking modes, variable-sized pages, and arbitrary reference patterns (i.e., graphs). This is achieved by relying on two widely-available OS primitives: anonymous memory mappings and an explicit memory-release system call. Although vmcache is a practical and useful design, with some workloads it can run into OS kernel performance problems. In this section, we describe a Linux kernel extension called exmap that solves this weakness. We first motivate why the existing OS implementation is not always sufficient, then provide a high-level overview of the design, and finally describe implementation details.

4.1 Motivation

Why Change the OS? With vmcache, (de)allocating 4 KB pages is as frequent as page misses and evict operations, i.e., the OS' memory subsystem becomes part of the hot path in out-of-memory workloads. Unfortunately, Linux' implementation of page allocation and deallocation does not scale. As a consequence, workloads that have a high page turn-over rate can become bottlenecked by the OS's virtual memory subsystems rather than the storage device. To quantify the situation on Linux, we allocate pages on a single anonymous mapping by triggering a page fault and evict them again with `MADV_DONTNEED`. As Figure 3 shows, vanilla Linux only achieves 1.51M OP/s with 128 threads. Incidentally, a single modern PCIe 4.0 SSD can achieve 1.5M random 4KB reads per second [4]. In other words, a 128-thread CPU would be completely busy manipulating virtual memory for one SSD – not leaving any CPU cycles for actual work.

Problem 1: TLB shootdowns. To investigate this poor scalability, we used the `perf` profiling tool and show a flame graph [17] in Figure 4. Linux spends 79% of all CPU time in the `flush_tlb_mm_range` function. It implements TLB shootdowns, which are an explicit coherency measure that prevents outdated TLB entries, which otherwise could lead to data inconsistencies or security problems. On changing the page table, the OS sends an *interprocessor interrupt (IPI)* to all other (N-1) cores running application threads, which then clear their TLB. This is fundamentally unscalable as it requires N-1 IPIs for *every* evicted page.

Problem 2: Page allocation. After shootdowns, the next major performance problem in Linux is the intra-kernel page allocator (*free pages* and *alloc page* in the flame graph). The Linux page allocator relies on a centralized, unscalable data structure and, for security reasons, has to zero out each page after eviction. Therefore, once the larger TLB shootdown bottleneck is solved, workloads with high page turn-over rates will be bound by the page allocator.

Why a New Page Table Manipulation API? The two performance problems described above cannot be solved by some low-level changes within Linux, but are fundamentally caused by the existing decades-old virtual memory API and semantics: The TLB shootdowns are unavoidable with a synchronous page-at-a-time API, and page allocation is slowed down by the fact that physical memory pages can be shared between different user processes. Achieving efficient and scalable page table manipulation therefore requires a different virtual memory API and modified semantics.

4.2 Design Principles

exmap. exmap is a specialized Linux kernel extension that enables fast and scalable page table manipulation through a new API and efficient kernel-level implementation. We co-designed exmap for use with vmcache, but as we discuss in Section 4.5, it could also be used to accelerate other applications. exmap comes as a Linux kernel module that the user can load into any recent Linux kernel without rebooting. Like the POSIX interface, exmap provides primitives for setting up virtual memory, allocating, and freeing pages. However, as outlined below, exmap has new semantics to eliminate the bottlenecks provoked by the POSIX interface.

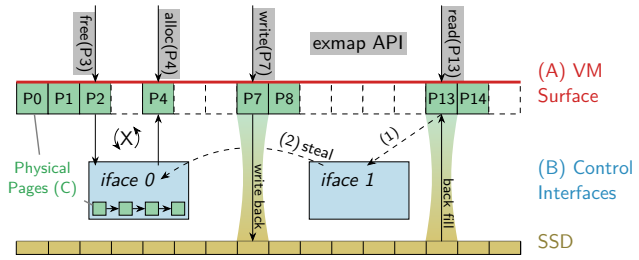


Figure 5: exmap implementation overview: The VM Surface (A) is manipulated with explicit free, alloc, read, or write system calls. Each per-thread control interface (B) owns part of the exmap-local memory pool, which exists as interface-local free lists of physical pages (C). If an interface runs out of pages (1), it steals pages from another interface (2). Pages only circulate (X) between the surface and the interface.

Solving TLB shootdown problem. An effective way of reducing the number of TLB shootdowns is to batch multiple page evictions and thereby reduce the number of shootdowns by the batch size. To achieve this, exmap provides a batching interface to free multiple pages with a single system call. While batching is easy to exploit for a buffer manager when evicting pages, it can be problematic to batch page allocations because these are often latency critical. To avoid TLB shootdowns on allocation, exmap therefore ensures that allocation does not require shootdowns at all. To do this, exmap always read-protects the page table entry of a freed page (by setting a specific bit in the page table entry). Linux, in contrast, sets that entry to a write but not read-protected zero page – potentially causing invalid TLB entries that have to be explicitly invalidated on allocation. This subtle change eliminates the need for shootdowns on allocation completely.

Solving the page allocation problem. Another important difference between `Linux` and exmap is the page allocation mechanism. In Linux, when a page is freed, it is returned to a system-wide pool (and thereby potentially to other processes). This has two drawbacks: (1) page allocation does not scale well and (2) pages are repeatedly zeroed out for security reasons. exmap, in contrast, preallocates physical memory at creation and keeps them in scalable thread-local memory pools – thereby avoiding both bottlenecks.

4.3 Overview and Usage

Implementation overview. Figure 5 illustrates the three major components of an exmap object: (A) its *surface* within the *virtual memory (VM)*; (B) a number of *control interfaces* to interact with the object; and (C) a private *memory pool* of physical DRAM pages, which exists as interface-local free lists spread over all interfaces.

Creation. On creation (lines 4-8 in Listing 3), the user configures these components: She specifies the number of interfaces that the kernel should allocate (line 5). Usually, each thread should use its own interface (e.g., thread id = interface id) to maximize scalability. The user also specifies the number of memory pool pages (line 6), which exmap will drain from Linux' page allocator for the lifetime of the exmap object. As the third parameter, the user *can* specify a file descriptor as backing storage for read operations (line 7).

```
1  // Open device/file as backing storage
2  int fd = open("/dev/...", O_RWDR|O_DIRECT);
3  // Create a new exmap object
4  struct exmap_setup_params params = {
5    .max_interfaces = 8, // # of control interfaces
6    .pool_size = 262144, // # of pages in pool (1 GB)
7    .backing_fd = fd}; // storage device
8  int exmap_fd = exmap_create(&params);
9  // Make the exmap visible in the VM
10 Page* pages = (Page*)mmap(vmSize, exmap_fd, ...);
11 // Allocate and evict memory using interface 5
12 exmap_interface_t iface = 5;
13 // Scattered I/O Vector: P1, P3-P5
14 struct iovec vec[] = {
15   { .iov_base = &pages[1], .iov_len = pageSize },
16   { .iov_base = &pages[3], .iov_len = pageSize * 3}};
17 exmap_action(exmap_fd, iface, EXMAP_ALLOC, &vec, 2);
18 exmap_action(exmap_fd, iface, EXMAP_FREE, &vec, 2);
19 // Read pages from fd into the exmap
20 // Use exmap_fd as a proxy file descriptor.
21 pread(exmap_fd, &pages[13], pageSize, iface); // P13
22 preadv(exmap_fd, &vec, 2, iface); // P1, P3-P5
23 // Write-Backs are explicit and without proxy fd
24 pwrite(fd, &pages[7], pageSize, 7 * pageSize); // P7
```

Listing 3: exmap usage example

Operations. After creation, the process makes the exmap surface visible within its VM via `mmap` (line 10). While an exmap can have an arbitrary VM extent, it can be mapped exactly once in the whole system. On the mapped surface, we allow the vectorized and scattered allocation of pages on the exmap surface (line 11 and Figure 5 (X)). For this, one specifies a vector of page ranges within the mapped surface and issues an `EXMAP_ALLOC` command at an explicitly-addressed interface. The required physical pages are first drawn from the specified interface (Figure 5 (1)), before we steal memory from other interfaces (Figure 5 (2)). Once allocated, pages are never swapped out and, therefore, accesses will never lead to a page fault, providing deterministic access times. With the free operation (line 18), we free the page ranges and release the removed physical pages to the specified interface.

Read I/O. In contrast to file-backed `mmap`, we do not page in or write back data transparently, but the user (e.g., vmcache) *explicitly* invokes read and write operations on the surface. To speed up these operations, we integrated exmap with the regular Linux I/O subsystem, whereby an exmap file descriptor becomes a proxy for the specified backing device (lines 19-22). This allows combining page allocation and read operations in a single system call: On read, exmap first populates the specified page range with memory before it uses the regular Linux VFS interface to perform the actual read. Since we derive the disk offset from the on-surface offset, we can use `offset` parameter to specify the allocation interface. With this integration, exmap supports synchronous (`pread`) and asynchronous (`libaio` and `io_uring`) reads. Furthermore, as the on-surface offset determines the disk offset, vectorized reads (`preadv`, `IORING_OP_READV`) implicitly become scattered operations (line 22), which Linux currently allows with no other system call.

Write I/O. On the write side, we actively decided against a write-proxy interface, which would, for example, bundle the write back and page evict. While such a bundling is not necessary as the user can already write surface pages to disk (line 24), freeing pages for each write individually could, if not used correctly, lead to unnecessary overheads. Therefore, we decoupled write back and (batched) freeing of pages.

4.4 Implementation Details

Scalable page allocator. Usually, when the kernel unmaps a page, it returns the page to the system-wide buddy allocator, which possibly merges it into larger chunks of physical memory. On allocation, these chunks are broken down again into pages, which have to be zeroed before mapping them to the user space. Therefore, with a high VM turn-over rate, memory is constantly zeroed and circles between the VM subsystem and the buddy allocator. To optimize VM operations for vmcache, we decided to use per-exmap memory pools to bypass the system allocator. This also allows us to avoid proactive page zeroing since pages only circulate between the surface and the memory pool within the same process, whereby information leakage to other processes is impossible. Only during the initial exmap creation, we zero the pages in our memory pool.
Thread-local control interfaces and page stealing. Furthermore, exmap's control interfaces not only allow the application to express allocation/eviction locality, but they also reduce contention and false sharing that come with a centralized allocator. For this, we distribute the memory pool as local lists of free 4 KB pages over the interfaces, whereby the need for page stealing comes up. After the interface-local free list is drained, we use a three-tiered *page-stealing strategy*: (1) steal from the interface from which we have successfully stolen the last time, (2) randomly select two interfaces and steal from the interface with more free pages, and (3) iterate over all interfaces until we have gathered enough pages. To minimize the number of steal operations, we steal more pages than required for the current operation. If we remove pages from the surface, we always push them to the specified interface. Thereby, for workloads in which per-interface allocation and eviction are in balance, steal operations are rarely necessary.
Lock-free page-table manipulation. For page-table manipulations, Linux uses a fine-grained locking scheme that locks the last level of the page table tree to update page-table entries therein. However, such entries have machine-word size on most architectures, and we can update them directly with atomic instructions. While Linux leaves this opportunity open for portability reasons, we integrated an *atomic-exchange–based hot path*: If an operation manipulates only an individual page-table entry on a last-level page table, we install (or remove) the VM mapping with a single compare-and-exchange.
I/O subsystem integration. For read operations, the Linux I/O subsystem is optimized for sequential reads into destination buffers that are already populated with physical memory. For example, without exmap, Linux does not provide a scattered read operation that takes multiple offsets; such a read request had to be split into multiple (unrelated) reads. On a lower level, Linux expects VM to be populated and calls the page-fault handler for each missing page before issuing the actual device operation. Hence, Linux cannot fully exploit scattered request patterns, but it handles them as individual requests which provokes unnecessary overheads (i.e., repeated page-table locking, allocator invocations). To avoid this, exmap provides vectorized and scattered reads with the proxy file descriptor. This allows us to (1) pre-populate the VM with memory, which avoids the page-fault handler path, and (2) cuts down the system-call overhead as we issue only a single system call per request batch.
Multiple exmaps. A process can create multiple exmap objects, which are mapped as separate non-overlapping *virtual-memory areas (VMAs)* into the process address space. These VMAs come with their own VM subsystem and are largely isolated from each other and from the rest of the kernel while still ensuring consistency and privilege isolation. As already noted, each exmap can be mapped exactly once, whereby we avoid the bookkeeping overhead of general-purpose solutions[5].

4.5 Discussion

OS customization. exmap is a new low-level OS interface for manipulating virtual memory efficiently. Seemingly minor semantic changes such as batching and avoiding zero pages result in very high performance without sacrificing security. One analogy is that exmap is for VM what `O_DIRECT` is for I/O: a specialized tool for systems that want to manage and control hardware resources themselves as efficiently as possible. Two design decisions of exmap require further discussion.
Functionality. We largely decoupled the exmap surface and its memory pool from the rest of Linux. As a consequence of this lean design, exmap is efficient but does not support copy-on-write forking and swapping. Few buffer pool implementations rely on such functionality. Indeed, it is actually a benefit that exmap behavior is simple and predictable as it allows buffer managers to precisely track memory consumption and ensure robust performance.
Portability. Another important aspect is generalizability to other operating systems and architectures. Since our kernel module comes with its own specialized VM subsystem, it only has few dependencies to the rest of the Linux kernel. This makes exmap easily portable between Linux versions and suggests that the concept can be implemented for other operating systems such as Windows and FreeBSD. Except for our architecture-dependent lock-free short-cut for small page table modifications, the exmap implementation is also independent of the used ISA and MMU as it reuses Linux' MMU abstractions. In other words, our Linux implementation is easily portable across CPU architectures that support Linux.
Other Applications of exmap. Although we explicitly designed exmap for caching, it has other use cases as well: (1) Due to its high VM-modification performance (see Figure 8), an heap manager could use a large exmap surface to coalesce free pages into large contiguous buffers, which is useful for DBMS query processing [14]. (2) With a page-move extension, a language run-time system could use exmap as a base for a copying garbage collector for pools of page-aligned objects. (3) For large-scale graph processing, workers request, often with a high fan out (e.g., for breadth-first search) and with high parallelism, randomly-placed data from the backing store, which can easily be serviced by exmap. (4) For user-space file

[5]For example, Linux usually maintains a reverse mapping from physical to virtual addresses that is necessary to implement features such as copy-on-write fork.

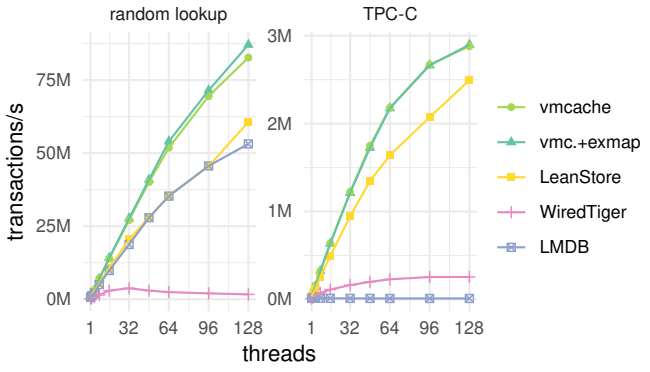


Figure 6: In-memory scalability (128 GB pool, random lookup: 100 M entries ≈ 20 GB, TPC-C: 200 warehouses ≈ 40 GB)

systems, a device-baked exmap allows for a user-space–controlled buffer cache strategy.

5 EVALUATION

The goal of this section is to show experimentally that vmcache is competitive to state-of-art swizzling-based buffer managers for in-memory workloads and that exmap enables the vmcache design to exploit modern storage devices. However, let us emphasize here that we see the main benefits of vmcache as qualitative rather than quantitative as we summarized in Table 1. Specifically, despite being easy to implement, vmcache supports arbitrary (graph) data and variable-size pages.

5.1 Experimental Setup

Implementation. Our buffer manager is implemented in C++ and uses a B+tree with variable-size keys/payloads and optimistic lock coupling. We compare two variants: (1) *vmcache* uses the regular and unmodified OS primitives described in Section 3. (2) *vmcache+exmap* is based on the vmcache code, except that it uses the exmap kernel module and the interface proposed in Section 4. Both variants use 4 KB pages and perform reads through the blocking `pread` system call. Therefore, there is at most one outstanding read I/O operation per thread. Dirty pages are written in batches of up to 64 pages using `libaio`. *vmcache* frees those pages individually with `madvise`, while *vmcache+exmap* batches them into a single `EXMAP_FREE` call. Page allocations are not batched to avoid increasing latencies (we use one `EXMAP_ALLOC` call per allocation).
Competitors. We use three state-of-art open source storage engines based on B+trees as competitors: (1) *LeanStore* [1], (2) *WiredTiger* 3.2.1 [5], and (3) *LMDB* 0.9.24 [2]. For caching, LeanStore and WiredTiger rely on pointer swizzling, whereas LMDB [2] uses `mmap` with out-of-place writes. Since the focus of this work is buffer management, in all systems we disable write ahead logging and run in the lowest transactional isolation level offered. LMDB and LeanStore use 4 KB pages, whereas WiredTiger uses 32 KB pages for leaf nodes on storage. We configured LeanStore to use 8 page provider threads that handle page replacement [19], which resulted in the best performance.
Hardware, OS. We ran all experiments on a single-socket server with an AMD EPYC 7713 processor (64 cores, 128 hardware threads) and 512 GB main memory, of which we use 128 GB for caching. For storage, we use a 3.8 TB Samsung PM1733 SSD. The system is running unmodified Linux 5.16, except when we run *vmcache+exmap*, which uses our exmap kernel module. **Workloads.** We use TPC-C as well as a key/value workload that consists of random point lookups, 8 byte uniformly-distributed keys, and 120 byte values. The two benchmarks are obviously very different from each other: TPC-C combines complex access patterns and is write-heavy, while the lookup benchmark is simple and read-only. Both are implemented as standalone C++ programs linked against the storage engines, i.e., there is no network overhead.

5.2 End-To-End In-Memory Comparison

vmache performance and scalability. In the first experiment we investigate the performance and scalability in situations where the data set fits into main memory. The results are shown in Figure 6. The two vmcache approaches are faster than the other systems and scale very well – achieving almost 90 M lookups/s and around 3 M TPC-C transactions/s respectively. Because no page eviction happens for in-memory workloads, we see that exmap does not offer major performance benefits over the basic vmcache design.
Competitor performance. LeanStore comes closest to vmcache in performance, while WiredTiger trails significantly. LMDB is competitive to LeanStore for the lookup benchmark but does not scale on the write-heavy TPC-C benchmark. This is because LMDB uses a single writer model with out-of-place writes, which means that reads do not have to synchronize, but only a single writer is admitted at any point in time. Overall, the results show that the vmcache design has excellent scalability and high absolute performance.

5.3 End-To-End Out-of-Memory Comparison

Workload. Figure 7 shows the out-of-memory performance (upper plot) over time. In this experiment, the data sets are larger than the buffer pool by one order of magnitude, which means that page misses happen frequently. We start measuring right after loading the data for both workloads. Therefore, in all systems it takes some time for the performance to converge to the steady state because the buffer pool state needs to adjust to the switch from loading to the actual workload.
vmcache and exmap. For the random lookup benchmark, we see that exmap improves performance over basic vmcache by about 60%. This is caused by Linux scalability issues during page eviction. For TPC-C, the difference between the vmcache and vmache+exmap is small because even vmcache manages to become I/O bound. For both workloads, the exmap variant manages to become fully I/O bound, as is illustrated by the lower part of the figure[6].
LeanStore. When we compare LeanStore with vmcache and exmap, we see that vmcache is substantially slower than LeanStore for random lookups in steady state (again due to vmcache being bound by

[6]We measured the I/O bound for this experiment using the `fio` benchmarking and 128 threads doing synchronous random I/O operations.

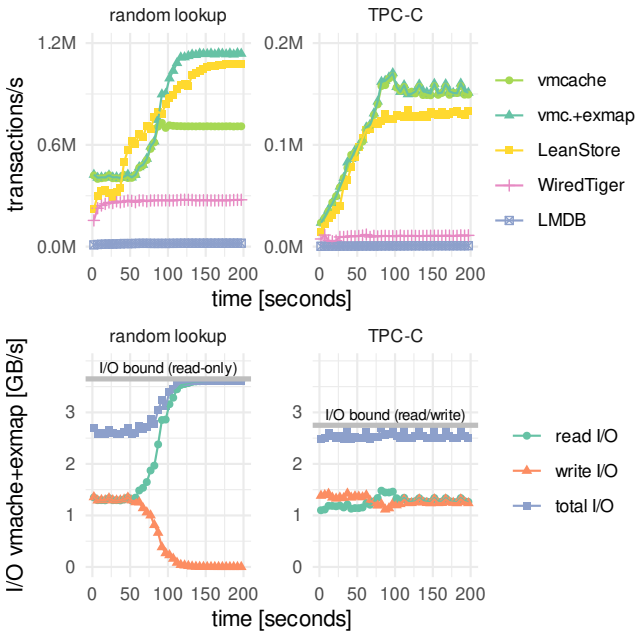


Figure 7: Out-of-memory performance and I/O statistics (128 GB buffer pool, 128 threads, random lookup: 5 B entries ≈ 1 TB, TPC-C: 5000 warehouses ≈ 1 TB)

the kernel). Only by using the exmap module, can it become competitive to LeanStore. Eventually, exmap+vmcache performs similarly to LeanStore and both become I/O bound in steady state. The performance differences are largely due to minor implementation differences: vmcache+exmap has slightly higher steady state performance due to a more compact B-tree (less I/O per transaction), and LeanStore temporarily (40s to 90s) outperforms vmcache+exmap due to more aggressive dirty page eviction using dedicated background threads.

WiredTiger and LMDB. WiredTiger and the `mmap`-based LMDB are significantly slower than vmcache and LeanStore. The performance of WiredTiger suffers from the 32 KB page size, whereas LMDB is bound by kernel overhead (random lookups) and the single-writer model (TPC-C). Overall, we see that while basic vmcache offers solid out-of-memory performance, as the number I/O operations per second increases it requires the help of exmap to unlock the full potential of fast storage devices.

5.4 vmcache Ablation Study

To better understand the performance of virtual-memory assisted buffer management and compare it against a hash table-based design, we evaluated page access time using a microbenchmark. We focus on the in-memory case, which is why all page accesses in this experiment are hits. For all designs, we read random 4 KB pages of main memory and report the average number of instructions, cache misses, and the access latency. We report numbers of 32 KB and 128 GB of data. The former corresponds to very hot CPU-cache resident pages and the latter to colder pages in DRAM. Line #1 in

Table 2: Random page access microbenchmark

		32 KB			128 GB		
#		inst-truc.	cache miss	time [ns]	ins-truc.	cache miss	time [ns]
1	read	3.0	0	1.6	3.3	1.0	219
2.1	read (1 TB range)	3.0	0	1.6	3.3	1.0	235
2.2	+ page state	7.0	0	1.7	7.4	2.0	236
2.3	**+ version check**	**10.0**	**0**	**1.8**	**10.4**	**2.0**	**236**
3	hash table	26.1	0	10.7	27.9	2.6	336

Table 2 simply shows the random access time in an 32 KB/128 GB array and therefore represents the lower bound for any buffer manager design. The next three lines incrementally show the steps (described in Section 3.1, Section 3.2, and Section 3.3) necessary in the vmcache design. In line #2.1, we randomly read from a virtual memory range of 1 TB (instead of 128 GB), which increases latency by 7% due to additional TLB pressure. In line #2.2, in addition to accessing the pages themselves, we also access the page state array as is required by the vmcache design. As mentioned in Section 3.6, this additional cache miss does not noticeably increase access latency because both memory accesses are independent and the CPU therefore performs them in parallel. In line #2.3, we also include the version validation, which results in the full vmcache page access logic. Overall, this experiment shows that a full optimistic read in vmcache incurs less than 8% overhead in comparison with a simple random memory read. We measured that an exclusive, uncontended page access (fix & unfix) on 128 GB of RAM takes 238ns (not shown in the table). The last line in the table shows the performance of a hash table-based implementation based on open addressing. Even such a fast hash table results in substantially higher latencies because the page pointer is only obtained after the hash table lookup. Note that our hash table implementation is not synchronized, and the shown overhead is therefore actually a lower bound for the true cost of any hash table based design.

5.5 exmap Allocation Performance

Allocation benchmark. The end-to-end results presented so far have shown that exmap is more efficient than the standard Linux page table manipulation primitives. However, because we were I/O bound, we have yet to evaluate how fast exmap actually is. To quantify the performance of exmap, we used similar allocation benchmark scenarios as in Figure 3, i.e., we constantly allocate and free pages in batches.

Baselines. The results are shown in Figure 8. For these, we always use batched allocations/evictions of 512 individual 4 KB pages. As a baseline, we use `process_madvise` with TLB batching, which already requires kernel changes. For reference, we also show the maximal DRAM read rate, which we achieved using the `pmbw` benchmarking tool and 64 threads (144.56 GiB/s). If the OS provides memory faster than this threshold, we can be sure that memory allocation will not be the bottleneck.

Page stealing scenarios. exmap uses page stealing, and its performance therefore depends on the specific inter-thread allocation

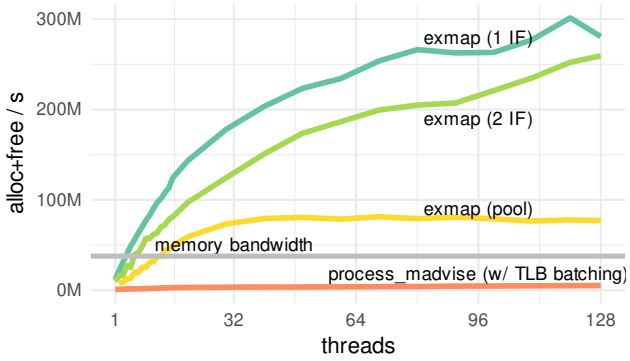


Figure 8: Linux memory allocation performance with exmap. The three exmap lines shows different page stealing scenarios (1 IF: no stealing, 2 IF: pair-wise stealing, pool: stealing across all threads)

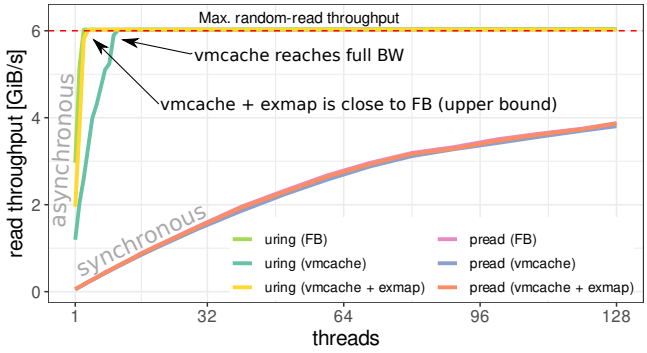


Figure 9: Read performance for synchronous (pread) and asynchronous (uring) I/O operations. Both vmcache and exmap support asynchronous I/O using uring, which allows achieving full I/O bandwidth using a few threads

pattern. We therefore investigate three workload scenarios at different degrees of page stealing: For *exmap (1 IF)*, no stealing occurs as each thread allocates 512 pages and then evicts them again at the same interface. *exmap (2 IF)* is like 1 IF but each thread has *two* interfaces, one for allocations and one for evicting pages. Due to a large enough memory pool (1 GB), stealing rarely occurs and we regularly steal more than 512 pages, but eventually each page must be stolen once per allocation. For *exmap (Pool)*, half of the threads allocate pages as fast as possible while the other half free those pages again. Here, the memory pool is always close to depletion and stealing happens frequently but often does not return more than 512 pages. Thereby, this scenario is the most challenging workload for exmap.

Results. Figure 8 shows that exmap outperforms the current state of the art in Linux significantly in all scenarios, and we reach up to 301M OP/s, which is equivalent of providing memory at 1,150 GiB/s and way beyond current DRAM speeds. We also see that page stealing has a moderate effect in the low-memory-pressure scenario (2 IF), while a high memory pressure (Pool) reduces the rate by 73 percent. This also demonstrates the success of our interface-local free lists and suggests that applications should try to roughly balance out their allocations and frees at each interface for optimal performance.

5.6 exmap Read I/O Performance

I/O libraries. Both vmcache and exmap support both synchronous (`pread`) and asynchronous I/O (`io_uring` and `libaio`). With asynchronous I/O, one can achieve high I/O using fewer threads. In the final experiment, we quantify the read-throughput of different user-space I/O strategies. For this, N threads randomly read 4 KB blocks in `O_DIRECT` mode from our Samsung PM1733 SSD with the synchronous `pread` system call and via Linux' modern asynchronous `io_uring` interface. The target memory is either vmcache, exmap, or thread-local fixed buffers (FB). As these FBs have a fixed address, this variant does not include VM-manipulation overheads and marks the upper bound of Linux' IO subsystem for the respective system-call interface. For the `io_uring` variant, we use thread-local submission queues and allow each thread to have 256 outstanding in-flight operations. We submit each read as an individual operation and do not use exmap's scattered and vectorized read capability. For vmcache, we use `process_madvise` with TLB batching for eviction, and for exmap, we read and evict at the same exmap interface. Since the SSD handles up to 128 parallel requests and has a maximum random-read throughput of 6 GiB/s, we are interested in which strategy can saturate it and how many threads it requires for this.

I/O performance. In Figure 9, we see that the `pread` variants, where each thread has at most one read operation in flight, cannot saturate the SSD. Nevertheless, both vmcache and exmap closely follow the throughput of the fixed-buffer variant, and we can conclude that our vmcache concept is not the limiting factor here. When using `io_uring`, where a single thread could already submit enough parallel reads to theoretically saturate the SSD, all three variants reach the maximum of 6 GiB/s at some point. With fixed buffers, 3 threads already saturate the SSD with 1.58 MIOP/s random reads. When using the regular Linux system-call interface to implement a vmcache, we require 11 threads to reach the same level. With a single thread, we reach 40 percent of the fixed-buffer performance. Even better, with exmap and `io_uring`, we only require 4 threads to reach 6 GiB/s and with three threads it is already at 96 percent. With a single thread, exmap achieves 66 percent of the single-threaded FB variant. We thus argue that in the modern hardware landscape in which multiple SSDs can be used, exmap is a perfect fit for buffer management. Both vmcache and exmap work with off-the-shelf asynchronous I/O in Linux. Furthermore, exmap minimizes virtual memory overhead and follows the performance of the upper-bound implementation (FB) very closely.

5.7 exmap Ablation Study

VM optimizations. Let us now quantify the impact of the exmap optimizations we presented in Section 4.4. For this, we perform an ablation study that is representative for scenarios with a high VM turn-over rate. The left-hand side of Figure 10 shows how the individual techniques contribute to exmap's performance. For the most

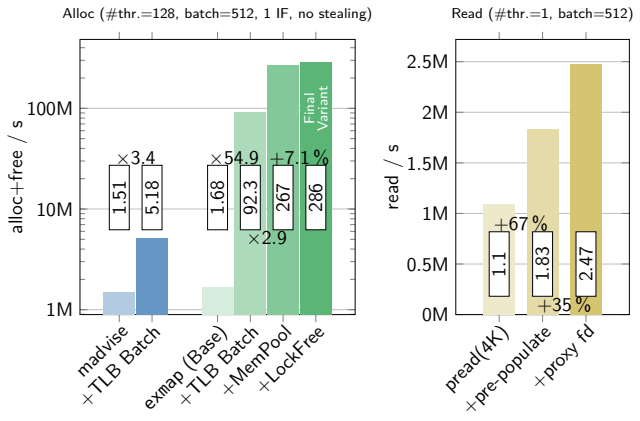


Figure 10: Impact of techniques on allocation (left) and read performance from `null_blk` (right).

basic variant (see Figure 10), exmap has a similar performance to `madvise`. However, for this variant, TLB shootdowns make up 98.16 percent of the total CPU time, which explains why TLB batching has a significantly higher impact on the end-to-end performance. With TLB batching, exmap reaches 92.34M OP/s. Our next optimization is to add the private scalable page allocator. With TLB batching and the private memory pool, exmap reaches 267.32M OP/s. Finally, we enable the lock-free page-table manipulation which further speeds up random page-sized surface manipulations and with this we reach 286.38M OP/s (see Figure 10). With our final variant of exmap, we outperform off-the-shelf `madvise` by a factor of ≈ 190.

I/O optimization. Our I/O-integration techniques contribute to exmap's performance as well. To quantify their contributions, we measure the read performance from a `null_blk` device (irqmode=0, queue_mode=0) onto an exmap surface. Due to scalability issues of the `null_blk` driver, we only show single-threaded performance. From the baseline, where reads are issued individually and provoke one page fault each, we first pre-populate the surface in batches of 512 pages and achieve 67 % more reads. Finally, by combining allocation with the actual I/O request through the proxy file descriptor, we gain another 35 % with batches of 512 scattered reads.

6 RELATED WORK

We already described prior work on buffer management in Section 2, so let us now discuss related work on virtual memory and operating systems.

Exploiting VM in DBMS. Besides caching, virtual memory manipulation has also been shown to be useful in other database use cases such as query processing [37], and for implementing dynamic data structures such as Packed Memory Arrays [26]. In multi-threaded situations, these applications may run into kernel scalability issues and would therefore likely benefit from the optimizations we propose in Section 4.

DBMS/OS co-design. Let us mention two recent DBMS/OS co-design projects. MxKernel [3] is a runtime system [32] for data-intensive systems on many-core CPUs. The focus of DBOS [38] is on cloud orchestration (i.e., managing and coordinating multiple instances) and on using database concepts and systems to simplify this task. Again, a technique like exmap is orthogonal to both designs and could be exploited by them.

Optimizing TLB shootdowns. The operating systems community has identified TLB shootdowns as a major performance problem and has proposed several techniques, including batching, for mitigating them [6, 7, 22]. exmap uses the same batching idea to speed up VM manipulation.

Incremental VM improvements. Existing work on improving the Linux VM subsystem can be split into two general categories: (1) speed up the existing infrastructure and (2) provide new VM management systems. For the first, Song et al. [39] modify the allocation strategy in the page fault handler. Freed pages are saved in application-local lists instead of being directly returned to the system, which enables the recycling of pages within an application. With exmaps, we extend this to explicitly-addressed free lists to avoid contention within the allocation path. Additionally, they batch write-back operations to mitigate the overhead of the write I/O path. Choi et al. [10] use cache removed VMAs for future use instead of deleting them immediately on `munmap`. They also extend the memory hinting system of `madvise`, adding new functionality like asynchronous map-ahead. Overall, the speedups of both these incremental approaches are limited because of the complex and general nature of the Linux VM subsystem. Another bottleneck of Linux' VM system is the management of the VMA list, which is a stored lock-protected red-black tree. *Bonsai* [11] uses an RCU-based binary tree to provide lock-free page faults. In follow-up work, *RadixVM* [12] speeds up mapping operations in non-overlapping address ranges. As exmap and vmcache only use a single long-living VMA and memory is not implicitly allocated through page faults, we do not expect significant speedups although they are orthogonal to our approach.

New VM subsystems. An alternative to incremental changes is to develop a specialized Linux VM subsystem. In UMap [35], memory-mapped I/O is handled entirely in user-space using *userfaultfd*. With memory hints for prefetching, caching and evicting, as well as configurable page sizes, they achieve a speedup of up to 2.5 times compared to unmodified Linux. UMap, similar to our exmap approach, manages separate regions that bypass the memory management of Linux. The approach also gives the application more control by providing configurable thresholds to influence the eviction strategy. Unlike vmcache, however, the kernel still controls page eviction. Furthermore, user-level page-fault handling introduces system-call overheads that run counter the goal of improving VM speeds. Papagiannis et al. identify bottlenecks in Linux's VM system and propose FastMap [34] as an `mmap` alternative for implicit memory-mapped file I/O. They alleviate lock contention through per-core free page lists as well as separate clean and dirty page trees. They also identify TLB invalidation as a limiting factor to scalability, which they also solve via batched TLB shootdowns. Overall, their implementation is up to 5 times faster than unmodified Linux, and provides up to 11.8 times more random IOPS/s. Though significantly faster than Linux' `mmap`, both UMap and FastMap offer no explicit control over page eviction, which makes them unattractive for database systems.

7 SUMMARY

vmcache. In this paper, we propose virtual-memory assisted, but DBMS-controlled buffer management. By exploiting virtual memory, vmcache is not only fast and scalable, but is also easy to implement, enables variable-size pages, and supports graph data. The basic vmcache design only relies on widely-available OS features and is therefore portable. This combination of features makes vmcache applicable to a wide variety of data management systems. vmcache is available at https://github.com/viktorleis/vmcache.

exmap. With fast storage devices, the page table manipulation primitives that vmcache relies on can become a performance bottleneck. To solve this problem, we propose exmap, a specialized OS interface for page table manipulation. We implemented exmap as a Linux kernel module that is highly efficient and scalable. When one combines vmcache with exmap, one can fully exploit even very fast storage devices. exmap is available at https://github.com/tuhhosg/exmap.

ACKNOWLEDGMENTS

The roots of this project lie in discussions at Dagstuhl Seminar 21283 "Data Structures for Modern Memory and Storage Hierarchies". This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 447457559, 468988364, 501887536.

REFERENCES

[1] 2022. LeanStore - A High-Performance Storage Engine for Modern Hardware. https://leanstore.io/.
[2] 2022. Lightning Memory-Mapped Database Manager (LMDB). http://www.lmdb.tech/doc/.
[3] 2022. MxKernel - A Bare-Metal Runtime System for Database Operations on Heterogeneous Many-Core Hardware. https://ess.cs.uos.de/research/projects/MxKernel/.
[4] 2022. Samsung PCIe Gen 4-enabled PM1733 SSD. https://semiconductor.samsung.com/ssd/enterprise-ssd/pm1733-pm1735/mzwlj3t8hbls-00007/.
[5] 2022. WiredTiger Storage Engine. https://docs.mongodb.com/manual/core/wiredtiger/.
[6] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *USENIX ATC*. 27–39.
[7] Nadav Amit, Amy Tai, and Michael Wei. 2020. Don't shoot down TLB shootdowns!. In *EuroSys*. 1–14.
[8] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *PVLDB* 14, 9 (2021), 1544–1556.
[9] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and robust latches for database systems. In *DaMoN*.
[10] Jungsik Choi, Jiwon Kim, and Hwansoo Han. 2017. Efficient Memory Mapped File I/O for In-Memory File Systems. In *HotStorage*.
[11] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable Address Spaces Using RCU Balanced Trees. In *ASPLOS*. 199–210.
[12] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2013. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *EuroSys*. 211–224.
[13] Andrew Crotty, Viktor Leis, and Andrew Pavlo. 2022. Are You Sure You Want to Use MMAP in Your Database Management System?. In *CIDR*.
[14] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. Experimental Study of Memory Allocation for High-Performance Query Processing. In *ADMS*. 1–9.
[15] Wolfgang Effelsberg and Theo Härder. 1984. Principles of Database Buffer Management. *ACM Trans. Database Syst.* 9, 4 (1984).
[16] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi A. Kuno, Joseph Tucek, Mark Lillibridge, and Alistair C. Veitch. 2014. In-Memory Performance for Big Data. *PVLDB* 8, 1 (2014), 37–48.
[17] Brendan Gregg. 2016. The flame graph. *Commun. ACM* 59, 6 (2016), 48–57.
[18] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*.
[19] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *SIGMOD*. 877–892.
[20] Alfons Kemper and Donald Kossmann. 1995. Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis. *VLDB Journal* 4, 3 (1995), 519–566.
[21] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *SIGMOD*. 691–706.
[22] Mohan Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselý, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. 2018. LATR: Lazy Translation Coherence. In *ASPLOS*. 651–664.
[23] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. 185–196.
[24] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019), 73–84.
[25] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *DaMoN*.
[26] Dean De Leo and Peter A. Boncz. 2019. Packed Memory Arrays - Rewired. In *ICDE*. 830–841.
[27] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*. 302–313.
[28] Gang Liu, Leying Chen, and Shimin Chen. 2021. Zen: a High-Throughput Log-Free OLTP Engine for Non-Volatile Main Memory. *PVLDB* 14, 5 (2021), 835–848.
[29] David B. Lomet. 2019. Data Caching Systems Win the Cost/Performance Game. *IEEE Data Eng. Bull.* 42, 1 (2019), 3–5.
[30] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *EuroSys*. 183–196.
[31] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distributed Syst.* 15, 6 (2004), 491–504.
[32] Jan Mühlig and Jens Teubner. 2021. MxTasks: How to Make Efficient Synchronization and Prefetching Easy. In *SIGMOD*. 1331–1344.
[33] Thomas Neumann and Michael Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.
[34] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2020. Optimizing Memory-mapped I/O for Fast Storage Devices. In *USENIX ATC*. 813–827.
[35] Ivy Peng, Marty McFadden, Eric Green, Keita Iwabuchi, Kai Wu, Dong Li, Roger Pearce, and Maya Gokhale. 2019. UMap: Enabling application-driven optimizations for page management. In *IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 71–78.
[36] Filip Pizlo. 2016. Locking in WebKit. https://webkit.org/blog/6161/locking-in-webkit/.
[37] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. 2016. RUMA has it: Rewired User-space Memory Access is Possible! *PVLDB* 9, 10 (2016), 768–779.
[38] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael J. Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2021. DBOS: A DBMS-oriented Operating System. *PVLDB* 15, 1 (2021), 21–30.
[39] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. 2016. Efficient memory-mapped I/O on fast storage device. *ACM Transactions on Storage (TOS)* 12, 4 (2016), 1–27.
[40] Michael Stonebraker. 1981. Operating System Support for Database Management. *Commun. ACM* 24, 7 (1981), 412–418.
[41] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *SIGMOD*. 1541–1555.
[42] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD*. 473–488.
[43] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *SIGMOD*. 2195–2207.