IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 7 November 2023, accepted 20 November 2023, date of publication 30 November 2023,
date of current version 11 December 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3338149

RESEARCH ARTICLE

Evaluation and Refinement of an Explicit Virtual-Memory Primitive

YANNICK LOECK AND CHRISTIAN DIETRICH
Operating System Group, Hamburg University of Technology, 21073 Hamburg, Germany
Corresponding author: Yannick Loeck (yannick.loeck@tuhh.de)

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 468988364, 501887536.
Publishing fees supported by Funding Programme Open Access Publishing of Hamburg University of Technology (TUHH).

ABSTRACT The general-purpose virtual-memory subsystem of Linux does not scale to the multi-million IOP/s SSDs of today. Further, its implicit write-back and back-fill semantic hinders data-intensive applications, like databases, more than it actually helps. In contrast, explicit virtual-memory abstractions, as introduced by ExMap, provide an Exokernel-like interface to manipulate the VM surface. Thereby, the user application remains in full control over the VM surface and can benefit from low operation latencies. In this paper, we investigate the performance of the existing ExMap abstraction and propose enhancements for its system-call interface, its process-local memory pool management, and the user-space page-state tracking. With memory-mapped I/O vectors, the global bundle list, and exported page tables, we improve the end-to-end throughput of in-memory random B+tree lookups by up to 5.7 percent, while the out-of-memory case improves by up to 4.9 percent.

INDEX TERMS Virtual memory, explicit file-mapped I/O, high-performance I/O, user-space solution, buffer manager, performance optimization.

I. INTRODUCTION

With modern SSDs alleviating the performance penalty for random access, and high-bandwidth memory [1] increasing the in-core processing speed, the *operating system (OS)*'s virtual-memory and I/O subsystems now have to handle multiple million events per second. For example, *one* Samsung PM1733/35 [2] SSD alone serves up to 1.5 million random 4 KiB reads per second (rand: 6 GiB/s, seq: 7 GiB/s). And while kernel-bypass technologies, like SPDK [3], bring high-performance I/O to the user space, the *virtual memory (VM)* subsystem is still a major bottleneck [4], [5], [6], [7].

For data-intensive applications, like *database management systems (DBMSs)*, this evolution is challenging and, together with the need for explicit control over the data [8], [9], forces the developer to leave aside existing OS abstractions in favor of user-space solutions. For the DBMS community, the buffer manager [10], [11], which is essentially a user-space equivalent of a page cache, is a seminal example of this approach. Such buffer managers require user-space page-state tracking and induce a page-lookup overhead for every access (e.g., via a hash table [12]), which slows down the latency-critical in-memory access path.

Therefore, Leis et al. [7] recently proposed VmCache, a new DBMS buffer-manager design that uses hardware-supported virtual memory to translate page identifiers to virtual-memory addresses, while retaining *explicit* control over back-fill/write-back decisions. In a nutshell, VmCache implements an explicitly-controlled file-mapped I/O abstraction: On top of a sparsely-populated anonymous mapping, whose size matches the storage device, VmCache explicitly reads/writes pages from/to disk at linearly-displaced mapping offsets. Thereby, VmCache significantly speeds up the in- and out-of-memory access paths and even outperforms the LeanStore buffer manager [11].

For VM manipulations, VmCache comes with the ExMap Linux kernel extension, which is a specialized *virtual address space (VAS)* abstraction for high-performance VM primitives. ExMap optimizes VM operations by (1) bypassing most parts of Linux's VM subsystem, (2) using process-local memory pools, and (3) providing batched operations, which avoid costly TLB shootdowns [13], [14]. Thereby, ExMap

The associate editor coordinating the review of this manuscript and approving it for publication was Wei Liu.

© 2023 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

addresses the performance limitations of VM under Linux and increases the performance of VmCache for an out-of-memory random B+tree lookup by 59 percent [7]. And although Leis et al. [7] presented ExMap as a vehicle to speed up VmCache operations, we believe that explicit virtual-memory management is a promising direction to extend the VM subsystem with specialized interfaces for fast asynchronous I/O devices.

In this paper, we undertake a thorough analysis of ExMap's performance, targeting further performance enhancements and reducing its memory footprint. We investigate different user–kernel interfaces, optimize the memory-pool operations, and use existing paging data structures to make user-space page-state tracking more efficient. We claim the following contributions:

- We evaluate the performance of the existing ExMap abstraction, analyzing it down to the cache-miss level.
- We improve the system-call interface with a memory-mapped command vector and private file descriptors.
- We replace the stealing-based memory pool with a cache-friendly process-local free-memory list.
- We extend the explicit memory-management concept by exposing page-table entries to the application.

The rest of the paper is structured as follows: In Sec. II, we briefly describe the *existing* ExMap abstraction and its current implementation, before we propose different enhancements in Sec. III. In the evaluation (Sec. IV), we quantify and compare the performance of the original ExMap design and our refined variant with synthetic benchmarks that cover a wide range of application patterns, and an end-to-end benchmark using VmCache. We discuss our results (Sec. V) and the related work (Sec. VI) before we conclude in Sec. VII.

II. THE ExMap ABSTRACTION

ExMap [7][1] is a kernel extension that allows direct and efficient user-space control over file-mapped I/O and anonymous memory. ExMap consists of three components: (1) the VM *surface*, which the user manipulates with explicit operations via (2) multiple explicitly-addressed *control interfaces*, and (3) the ExMap-private *memory pool*, which contains physical page frames that back the surface *virtual-memory area (VMA)*. In contrast to classical file-mapped I/O, ExMap does *not* provide an implicit back-fill and write-back automatism but instead necessitates the explicit initiation of these operations; all page faults are forwarded as segmentation-fault signals to the user space.

A user-space application creates an ExMap object with a certain VM *surface* extent, and sets the number of physical page frames that back the *memory pool*. For the ExMap lifetime, these page frames are drained from the system allocator, and they are only given back to the system when the ExMap object is destroyed. As the ExMap and its memory pool are by design *process-private*, an ExMap instance cannot be shared with other processes but the OS is also relieved from page zeroing as data cannot leak into other processes.

After mapping the surface, the process uses *control interfaces* to issue commands that allocate/release pages or trigger I/O requests. For each application thread, ExMap uses a dedicated interface, establishing a one-to-one mapping between thread and interface. These interfaces are tightly coupled with the memory pool; all available page frames are stored in interface-local linked lists, and used for the requested operations. If the interface-local list does not hold enough frames, ExMap *steals* them from other interfaces. After gathering enough frames, ExMap manipulates the page tables of the surface VMA with atomic *compare-and-exchange* instructions. In contrast to CPU-local free lists, ExMap's interfaces allow for controlled locality and give the user space the possibility of limiting the dispersion of pages if fewer threads than cores are used.

Further, ExMap provides system calls for scattered and vectorized VM-surface operations, which also results in TLB-shootdown batching [13], [14], [15]. To specify multiple ranges, the user passes a vector of address–length pairs (i.e., `struct iovec[]`). Also, the ExMap file descriptor acts as a `proxy` for (a)synchronous read operations from the actual storage devices: With a single read system call, ExMap first maps the specific ranges before it forwards the request to the backing descriptor. For write operations, no such proxying exists.

A. INTEGRATION WITH VmCache

The VmCache buffer manager creates an ExMap instance with the extent of the underlying storage medium and manages the state of all pages in the user space (see Lst. 1). In its `page_state` array, which requires one element per disk page, VmCache tracks which pages are currently present on the surface to avoid the costly page-fault path. Further, the page state also holds a page's locking state (i.e., write-exclusive, read-shared) and a version number for optimistic reads.

In Lst. 1, the `fixEmpty()` function uses ExMap to allocate a fresh page at a given offset on the surface. For this, it uses the page-state array to check if the page is already present. If not, the application passes the on-surface offset to ExMap via the `ioctl()` syscall, which returns the amount of newly populated memory. If the allocation was successful, meaning that *this* operation inserted one page, we update the page state. Please note, that (a) the used `iovec` only has exactly one element, and (b) that the given example contains no race condition as from two concurrent invocations of `fixEmpty()` only one `ioctl()` invocation returns `PAGE_SIZE`;

Summarized, ExMap is a specialized kernel extension that provides fast VAS primitives to be used by an application that performs its own data management (e.g., DBMS buffer manager). In their experiments, Leis et al. [7] showed that ExMap is able to handle up to 286 million 4 KiB allocations

[1] ExMap is available at https://github.com/tuhhosg/exmap

IEEE Access

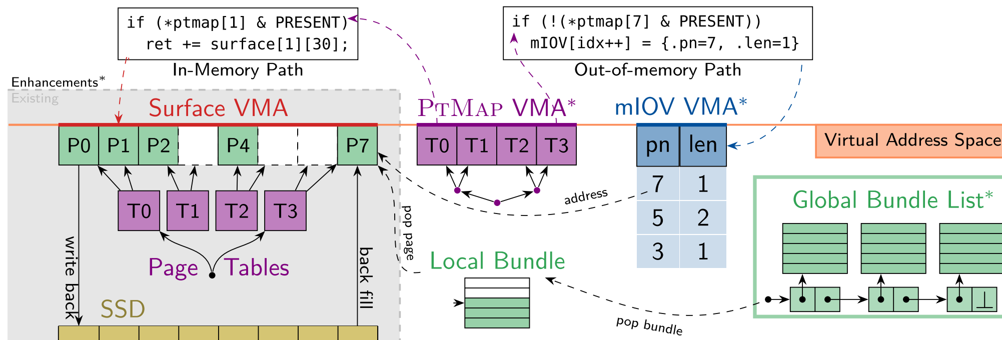


FIGURE 1. Design overview of the enhanced (*) ExMap Design: Within the virtual address space, an ExMap's explicitly-managed *surface* is mapped as a virtual-memory area (VMA). By exporting the page tables (PtMap VMA), the user space can detect if a page is present. The memory-mapped syscall interface (mIOV) allows the bulk transfer of system-call parameters. The ExMap's memory pool is managed with a lock-free global bundle list.

```
int exmap_fd = open("/dev/exmap", ...);
thread_local int exmap_iface = ...;
uint8_t *surface = mmap(exmap_fd,....);
uint8_t page_state[1 << 28]; // for 1 TiB surface

void* fixEmpty(PageNumber pn) {
  if (page_state[pn] == 0) { // page is not present
    // Allocate one page at offset
    uintptr_t offset = pn * PAGE_SIZE;
    struct iovec vec = {
      .iov_base = surface + offset,
      .iov_len  = PAGE_SIZE;
    };
    int rc = sys_ioctl(exmap_fd, exmap_iface,
                       /* op= */ EXMAP_OP_ALLOC,
                       /* vector = */ &vec,
                       /* length = */1));
    if (rc == PAGE_SIZE) {
      page_state[pn] = 1;
      return (void*)&surface[offset];
    }
  }
  // Page was already present
  return NULL;
}
```

LISTING. 1. Usage Example of ExMap. The `fixEmpty()` function ensures that the page at `pn` is present in memory, allocating a fresh memory page if necessary.

per second. In contrast, established Linux VM abstractions only achieve up to 5.2 million operations in this benchmark.

III. ExMap ASSESSMENT AND REFINEMENTS

In the following section, we evaluate the current version of ExMap and refine it in three architectural aspects: the system-call interface, the page-state tracking, and memory-pool management. Our goal is both to enhance ExMap's performance and to serve as a blueprint for improving other high-performance kernel extensions.

A. SYNCHRONOUS SYSTEM-CALL INTERFACES

The ExMap kernel extension is governed from the user space using a kernel interface. This necessitates finding the most suitable Linux interface or system call, factoring in characteristics like operation batching and invocation overhead. The aim is to establish a versatile solution to accelerate user-kernel communication for extensions that need both rapid synchronous operations (i.e., for allocation and freeing), which benefit from batching to avoid TLB shootdowns, and asynchronous operations (i.e., for I/O commands).

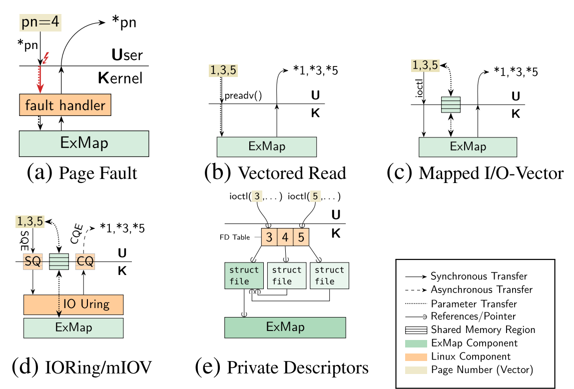


FIGURE 2. Different control interfaces for ExMap.

1) PARAMETER PASSING

Traditionally, applications request memory on a VMA via *page faults* (see Fig. 2a), making it an implicit synchronous control interface. However, besides known bottlenecks [7], [16], [17], page faults have the pitfall to require one kernel activation per requested frame as the principle does not allow for batching.

Therefore, ExMap already comes with a vectorized operation interface. Traditionally, for address-space–related operations, Unix uses a `struct iovec`-array to pass scattered VAS references to the kernel. For example, `readv`[2] (see Fig. 2b) reads sequentially from disk or file into a scattered buffer described by the `iov` elements. While ExMap already supports this interface for reading and manipulating at multiple surface offsets with one system call, it has the *return-value problem*. Conceptually, `readv` is one operation with one return value (number of read bytes), but batched operations consist of multiple independent operations, each

[2] `ssize_t readv(int fd, const struct iovec *iov, int iovcnt)`.

of which could fail independently. When using `readv` for ExMap, we therefore have to stop processing the vector at the first conflict/error (e.g., empty memory pool, modification conflict).

To address this problem, we extend each ExMap interface with a *mapped I/O vector* (mIOV) (see Fig. 2c), which is a permanently-present page for bi-directional parameter/return value transfer. This mIOV page has to be allocated per thread, which we do by tying it to ExMap interfaces.

Before invocation, the user encodes its I/O vector into the mIOV of the interface she wants to use. As surface operations are page aligned and likely reference only a few pages, we use a dense vector-element encoding with 52 bits for the page number and 12 bits for the length in pages. Thereby, the user can pass 512 independent operations on a single mIOV page to the kernel, without copying. On return, the kernel transfers the operation result via the same mIOV entry to the user.

mIOVs are similar to and inspired by `io_uring`'s [18] persistently-mapped command queues and *registered buffers*. With the recent introduction of `ucmd` (v5.19), `io_uring` gained an extensible interface for asynchronous commands that a file descriptor can provide. In combination with registered buffers, this allows us to extend ExMap with an interface similar to mIOV but more in line with existing Linux interfaces (see Fig. 2d). In Sec. IV-D, we will compare the different parameter passing variants.

2) INVOCATION

Like ExMap, many Linux interfaces use file descriptors as handles for non-file objects (e.g., `io_uring`, `signalfd`,...). And although accessing the file-descriptor table [19] is a scalable operation today, accessing the same file descriptor concurrently is still problematic: If 128 cores allocate individual pages (see Fig. 3) via ExMap, 81 percent of the time is used for reference counting (`fget/fput`), while the actual work load takes up only 4 percent.

To alleviate this false sharing without adapting basic Linux design decisions, we propose *private descriptors* (see Fig. 2e), which are clones of other file descriptors to be used by only one thread. They hold a reference to the original file descriptor and only forward the invoked operations. Thereby, in the example, a system call with `fd=3` and `fd=5` are semantically equivalent, but will access different reference counters. The registered-descriptor concept of `io_uring` solves the same problem, but is specific to `io_uring`, with an SQE flag indicating whether a registered or a normal file descriptor was given. In contrast, private descriptors are actual file descriptors and can be used with any existing system-call interface.

B. GLOBAL BUNDLE LIST–MEMORY-POOL MANAGEMENT

As every ExMap operation interacts with the *memory pool*, the efficient management of free frames is a core performance consideration. As described in Sec. II, ExMap currently stores its memory pool in interface-local free lists. These free lists are protected with locks due to the possibility of concurrent accesses during page stealing. With high memory-pressure situations, this can cause lock contention which negatively affects performance. Here, ExMap is currently overly complex in two regards: (1) it uses cache-inefficient doubly linked lists, and (2) the free memory is scattered over as many different locations as there are application threads.

Instead, we propose an architectural simplification of the memory pool, which also improves its performance (see Sec. IV-F): For that purpose, we introduce the *global bundle list (GBL)*, a lock-free singly linked list that is global for an ExMap object and whose elements are *bundles* of free frames. Instead of making the bundles linked lists themselves, we use the underlying memory of one frame as a stack that stores pointers to further free frames. Thus, in addition to the bundle-stack frame itself, a full bundle contains 512 more free frames if the system uses 4 KiB pages. With this, each GBL operation adds or removes 513 free page frames to/from the pool, which eliminates the need for page stealing from different interfaces. Additionally, bundles are more cache-friendly than normal lists, as one cache line worth of bundle-stack memory contains eight frames, while a linked list requires one cache line per element.

Making the GBL lock-free requires atomic compare-and-exchange based interaction for element addition and removal. For this, we rely on the widely-supported single-word *CAS* operation, necessitating the list to be singly linked. As atomic singly linked lists suffer from the ABA problem, we add a tag, which is incremented on pushes, to the first list entry. As frame addresses are 4 KiB-aligned, we can have 12 bits for the tag, which we consider sufficient to solve the ABA problem for our use case as our in-kernel operations are not subject to preemption.

While we replace stealing with the GBL, we keep the command interfaces but equip them with one *local bundle* as a free-page cache (see Fig. 1). This local bundle is taken from the global list and we keep it as long as it holds between 1 and 512 pages, where it satisfies allocation and free operations without the need to interact with other interfaces. When a page is freed and the local bundle is almost full (512 pages), we push the bundle to the GBL and the interface then has an empty cache. On the next free, we use the incoming page as the bundle-stack page of the next bundle. Only if a cache has no local bundle, we take one from the GBL.

Using local bundles means that per interface, up to 2 MiB of free memory is inaccessible for other threads as we do *not* perform stealing. However, if we keep a strict one-to-one mapping between threads and interfaces, less memory becomes inaccessible than with CPU-local free lists if an application has fewer threads than cores. Nevertheless, we consider 2 MiB of non-reclaimable memory per core as unproblematic.

With our GBL architecture, only one out of 513 page allocations has to interact with other CPUs, while the other 512 requests are serviced by the interface's local bundle.

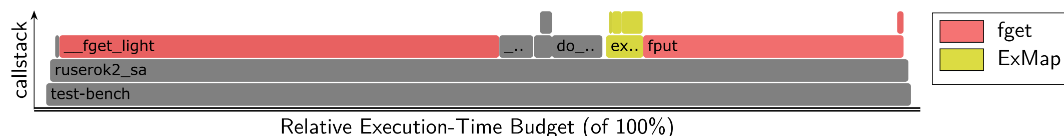


FIGURE 3. Flamegraph of single-page allocations with ExMap on 128 cores (11 M OP/s).

```
uint64_t getPageVersion(PageNumber pn) {
  if (ptmap[pn].present)
      return surface[pn * PAGE_SIZE].page_version;
  return VERSION_INVALID;
}
```

LISTING. 2. Page version for optimistic reads.

Furthermore, the interface-local page cache is refilled with a single lock-free pop operation instead of having to collect pages one by one.

C. PtMap–EXPORTED PAGE TABLES

With the transition from implicit to explicit VMA-surface management, the user requires the knowledge which page is currently mapped [7]. For example, in Fig. 1, P1 is mapped while P3 is currently not backed by physical memory (green). Leis et al. [7] combined user-space page-state tracking with locking and versioning. As they store an additional 64-bit word per page, the *page-state array* grows with the number of SSD pages. In comparison, traditional DBMS buffer managers only require space proportional to the number of used DRAM pages. Further, as file-mapped I/O requires more page tables, we need an additional ∼8 bytes for the *page-table entry (PTE)*; making VmCache more memory hungry than a traditional hash-table–based buffer manager.

To reduce this space overhead, we extend ExMap with PtMap, which exposes the last level of the page-table tree as a read-only mapping to the user space. Thereby, the user can directly query the `PRESENT` bit in the corresponding PTE (see Fig. 1) to check if a page is mapped. As this bit informs us whether an access will take the in- or the out-of-memory path, applications can use a less memory-intense schema to store per-page meta data. For example, we can store the rest of the page state (lock, version) within the mapped page itself. Further, exposing the PTEs also gives the user access to MMU-managed information about accessed and dirty states (on platforms where these are available). However, using this additional information is out of scope for this paper.

Technically, the PtMap is a second VMA that is much smaller than the surface VMA as it only has to make one PTE per page visible. On AMD64, the PtMap has a 512 times (4096 / 8) smaller extent, which also limits the MMU overhead for the PtMap VMA itself to $8/512$ bytes per page. Further, the PtMap maps the same physical memory that is also used for the surface page tables. Therefore, in Fig. 1, the four physical pages T0-T3 are not only the page tables of the surface VMA but also the user-readable memory of PtMap. Please note, that we populate the PtMap and surface VMAs lazily with page tables and frames, as it is common practice in Linux.

Sharing page-table data between surface and PtMap has two important consequences: (1) physical memory addresses of memory-page pools leak into the user space, which we will discuss from a security point of view in Sec. V. (2) PtMap and the ExMap surface are always in sync as ExMap also uses atomic operations to modify the page tables. Thereby, the present bit in the PTE is an atomic synchronization point for optimistic page accesses.

Another benefit of PtMap is cache related: As the MMU's page table walker also requests memory through the cache, it can reuse cache entries that were already fetched by the PtMap access. Please note, at page granularity, modern caches are physically-indexed to avoid the synonym problem [20]. In contrast, the externally-stored page-state array will provoke more cache misses on a TLB miss (see Sec. IV-E).

1) INTEGRATION WITH VmCache

As already mentioned, PtMap enables making user-space page-state tracking less memory intensive. For this, we strip the page-presence bit from the page state and store the remaining information (lock state, version tag) within the page itself. While this in-band page state increases the on-disk space requirement, it also allows us to get rid of the externally-stored page state array. As SSD storage is much cheaper than DRAM, this trade-off is worthwhile.

Within the access path (see Lst. 2), we access the PtMap to check if a page is present before accessing the surface. Please note, that VmCache already handles the situation where a concurrent eviction removes a page while another thread accesses it optimistically without a (reader) lock. Technically, this is done by translating segmentation-fault signals to C++ exceptions that abort the transaction.

IV. EVALUATION

The goal of our evaluation is to quantify and compare the performance of the original ExMap design with our enhanced variant. As VmCache's performance is currently primarily limited by memory bandwidth and I/O latencies, we first focus on a family of synthetic micro-benchmarks to allow a separate in-depth analysis of the ExMap design at its limits. Afterwards, we will perform an end-to-end evaluation to compare the original ExMap with our refined variant within the context of VmCache.

Here, we compete against already highly optimized baselines, with ExMap being able to serve 100-400 million allocation requests. In contrast, an optimized Linux variant, using system calls already patched for better performance, can only achieve 5 million allocations. Further, VmCache already efficiently utilizes ExMap to be 20 million transactions/s faster than the highly performant Lean Store engine for the in-memory case. For the out-of-memory case, which is instead dominated by the I/O bandwidth of the SSD device, our design nevertheless improves performance as it reduces the memory overhead which increases the usable memory available to the buffer manager. This shows that for both the synthetic benchmarks and the end-to-end evaluation, even improvements of a few percent over the original ExMap are significant.

A. EXPERIMENTAL SETUP

1) EXECUTION PLATFORM

We execute our experiments on a single-socket AMD EPYC 7713 server (Zen 3, 64 cores, 128 hardware threads) with 512 GiB of DRAM. Each core has 32 KiB of L1D cache and 512 KiB of L2 cache, while all cores share 256 MiB of L3 cache. As secondary storage options, we have eight 3.84 TiB Samsung PM1733 SSDs (PCIe Gen4, MZWLJ3T8HBLS), each of which is rated at 7000/3800 MiB/s read/write throughput and 1.5 M IOP/s. Our system runs on an unmodified Linux 6.1.8 kernel into which we load the respective ExMap variant as a loadable kernel module. We use the GNU C++ compiler (12.2.0) with `-O2` optimization level and jemalloc 5.2 for dynamic heap management, as it outperforms the standard GNU libc allocator [6].

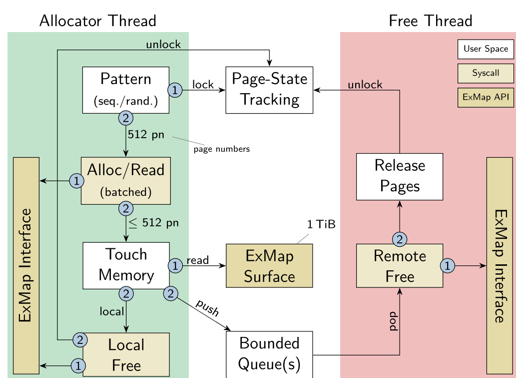


FIGURE 4. Benchmark overview. Allocator threads request pages on the ExMap surface, while free threads release pages with the remote and mixed free strategy.

2) SYNTHETIC BENCHMARKS

For our evaluation, we use a family of synthetic benchmarks that resemble different usage patterns of an explicit VAS abstraction. We do not only look at the out-of-memory I/O case, which are usually dominated by wait times, but also at CPU-bound use cases where ExMap is used for memory allocation. In order to push ExMap to its limits, we designed our benchmarks to have little (cache and performance) impact, which we will quantify in Sec. IV-B.

In Fig. 4, we show the common structure and operation of our synthetic-benchmark family: We use one ExMap instance with a 1 TiB surface and 128 MiB of DRAM in the memory pool per allocator thread. In order to mimic the situation in a steady state, we initialize ExMap's memory pool by allocating all pages and before freeing them randomly.

Furthermore, similar to [7], we use an externally-stored page-state array, which we manipulate concurrently and lock-free via atomic instructions, to track in-memory/allocated pages. As we use one byte per surface page, the array takes up 256 MiB of DRAM per TiB and alone would fill up our machine's L3 cache, making each random state access a likely cache miss.

Each *allocator thread* uses one of two *patterns* to select 512 page numbers, which are immediately locked in the page-state array. With the *sequential pattern*, the thread requests, in a circular fashion, pages from a 512 MiB-large thread-local address range; if previously requested pages are not freed yet the batch contains holes. This pattern mimics parallelized sequential-scan workloads. For the *random pattern*, the thread selects and locks 512 random pages from all over the ExMap surface; if a requested page is still locked, we choose another one. This pattern mimics workloads that make full use of the random-read capability of modern SSDs.

As a second step, the allocator thread requests those 512 pages via its thread-private ExMap control interface. For allocation scenarios, we batch all pages at once, while we issue 512 separate `io_uring` read SQEs with a single system call for SSD-read scenarios. Further, we try to keep 512 read requests in flight, which results, even if addressing 8 disks in parallel, in a queue length of 64, which is sufficient to saturate our SSDs [2]. As an allocation might fail in principle (e.g., memory shortage), the touch-memory stage accesses only one cache line from successfully allocated pages.

Afterwards, we use one of three free strategies to release pages: (1) With *local free*, the allocator thread uses its own ExMap interface to free the pages before unlocking them in the state array; this strategy mimics, for example, short-lived thread-local memory allocations. (2) With *remote free*, the allocator pushes the pages (as a batch) to a bounded queue, where a free thread takes over the responsibility, frees the pages, before it unlocks the page again; this strategy mimics the usage of a centralized buffer manager with a coordinated cache eviction strategy. (3) With the *mixed-free strategy*, the allocator releases half of the pages locally, while the other half is pushed to the remote-free queues.

Since we use the same number of allocator and free threads, we only state the number of allocator threads and report the number of allocation operations. Also, we use one bounded blocking queue per free thread to avoid lock contention, and the push operation tries four queues before it actually blocks. In sum, the combined capacity of all queues is half the size

of the memory pool (64 MiB per thread), whereby allocations cannot fail due to memory shortage.

During a benchmark run, we use `perf_event` [21] to record different CPU counters. We focus on last-level cache misses as the memory latency today dominates many CPU-bound workloads. To minimize the measurement impact, the benchmark reports its results only once per second; each benchmark configuration is run for 60 seconds.

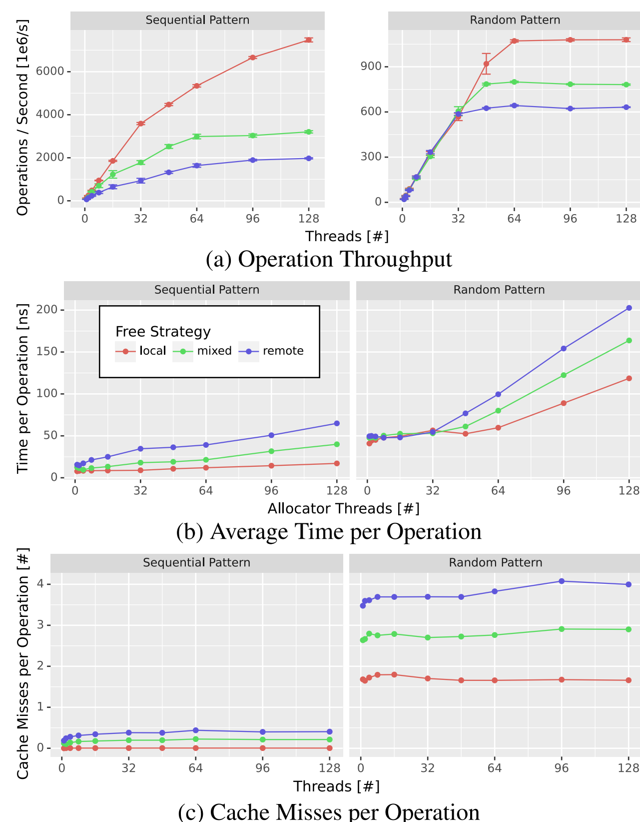


FIGURE 5. **Benchmark overhead *without* ExMap.**

B. BENCHMARK CHARACTERIZATION

ExMap was reported [7] to sustain hundreds of millions of surface operations per second, a scale at which every induced cache miss has a significant impact on the performance of in-memory workloads. This necessitates detailed evaluation and analysis, both to understand the measurement results and to discover further optimization potential. Simultaneously, it is important for the benchmark itself to have a minimal overhead, in order to provide an accurate evaluation of the different ExMap variants. Before analyzing those, we first characterize our synthetic benchmarks by quantitative measurements. This will not only help to understand the different variants, but we can also quantify the overheads, in terms of latency and cache pressure that is induced by our benchmark.

To this end, we execute our benchmarks (see Fig. 4) without performing the ExMap system calls. Therefore, only the pattern generation, the page-state manipulation, and the interaction between the allocator and free threads through the bounded queue remain. Given a theoretically ideal synthetic benchmark (perfect scalability, no cache impact), we could measure the upper possible bound of each ExMap variant.

We execute the six benchmark configurations (2 patterns × 3 free strategies) with up to 128 allocator threads, which is the number of hardware threads. However, we already reach one thread per core with 32 mixed and remote allocator threads, or with 64 local allocators.

From the results (see Fig. 5), we see that sequential/local is the best case as it runs purely from the cache as each thread only manipulates the same 128 KiB of the state array. Therefore, at 128 threads, each "allocation" takes 17 ns and provokes 0.004 (last-level) cache misses, which yields 7.48 billion operations per second as it boils down to setting/clearing bits on CPU-local L2 cache lines. However, even if we "free" pages remotely, we still reach 1.97 billion operations per second (65 ns each).

For the random pattern, we see a significant impact of the page-state tracking, as two randomly picked pages are unlikely ($2.61 \cdot 10^{-6}\,\%$) to be on the same cache line. Therefore, each operation will result in at least one cache miss if done purely locally, and two misses if done remotely. Nevertheless, in this scenario, the AMD Zen 3 cache prefetcher provokes twice as many cache misses as expected (Fig. 5c). To confirm this being a hardware anomaly and not caused by our benchmark, we executed the same benchmark on an AMD Ryzen 7 Pro 5850 CPU. There, the cache-miss/operation ratio was in the expected range (local: 1.06, remote: 2.25). Nevertheless, even in the worst-case (remote/random), our benchmark still reaches 631.7 million operations/s on the Zen3 machine.

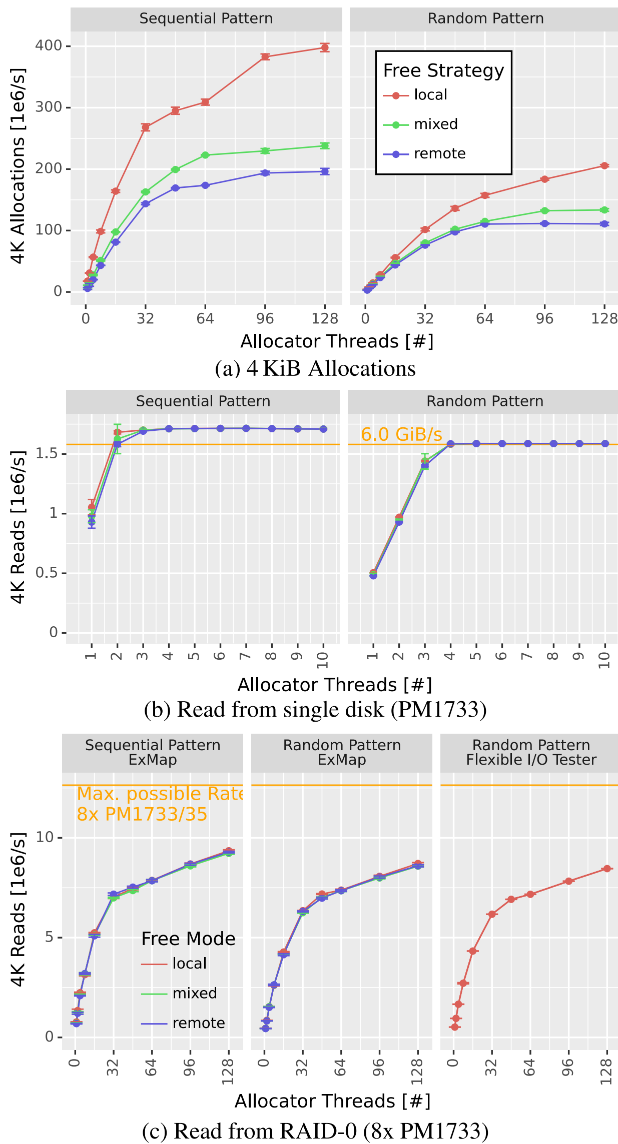
With our characterization, we show that sequential/local has an insignificant impact per operation and even the worst-case random/remote only induces four cache misses on our evaluation platform.

C. BASELINE EVALUATION

Next, we will look at the performance of the existing [7] ExMap abstraction and measure 4 KiB allocations as well as read requests on a single SSD and a Linux Software RAID-0 with eight SSDs. With the RAID measurement, we investigate whether Linux in combination with ExMap is able to saturate the theoretically available 12 million IOP/s.

1) ALLOCATIONS

In Fig. 6a, we confirm that ExMap is able to supply at least 100 million 4 KiB on its surface, even if those pages are selected randomly and allocated and freed by different threads. Even then, ExMap still provides 111M pages per second for 128 threads, which, if read entirely, would require 423 GiB/s DRAM bandwidth. At this point, an operation takes 1.16 us and induces 14.4 misses. Besides the 4 misses for the benchmark, this number includes frame allocation (with stealing), two lock-free page-table manipulations, one data access and the batched TLB shootdown.



(a) 4 KiB Allocations

(b) Read from single disk (PM1733)

(c) Read from RAID-0 (8x PM1733)

FIGURE 6. **Original ExMap performance.**

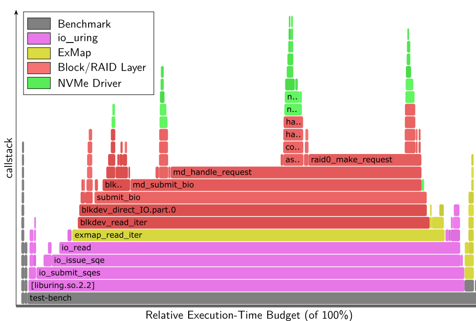


FIGURE 7. **ExMap flamegraph on RAID-0 (128 threads).**

2) SINGLE-DISK READ

In Fig. 6b, we look at the read performance of ExMap. For this, we use ExMap's combined alloc+read operation via the proxy descriptor. To our surprise, for the sequential pattern, we see that we achieve more than the 1.5M IOP/s (6 GiB/s) our PM1733 should provide. This is possible as the Linux block layer [22] merges requests if they are sequential on disk, even if issued as separate read requests (as our benchmark does). Thereby, we come close (6 677 MiB/s) to the 7000 MiB/s for sequential reads.

With the random pattern, we confirm that ExMap is able to saturate a PM1733 with four allocator threads. At this point, with random/remote, each operation requires 41 cache misses. We see that the achieved throughput does not depend on the free strategy.

3) RAID-0 READ

Fig. 6c shows the read throughput achieved with an 8-disk software–RAID-0 device. As both patterns hit at least 4 disks with a 512-page batch, our benchmark could, in principle, reach the full throughput of this setup with more than two threads. However, ExMap never reaches this maximal throughput, even if all 128 cores request blocks sequentially. As this does not meet our projections from the single-disk case, we execute a 4 KiB random-read benchmark with the flexible I/O tester [23] (fio) and its `io_uring` benchmark, which reads the data into fixed buffers instead of installing them in the VAS. However, fio achieves only 2 percent (geometric mean) more throughput than ExMap random/local; exonerating ExMap as the prime suspect.

To confirm this, we record a flamegraph [24], whose x-axis documents the relative execution-time budget and the y-axis the call stack. In Fig. 7, we categorize the different execution-time components: while ExMap is only in 7 percent of the samples on top of the call stack, the NVMe driver (11 %) and `io_uring` (13 %) require around twice as much execution-time budget. With 67 percent, the block layer, which performs the RAID-0 striping, the I/O scheduling, and NVMe request submission, is the biggest contributor to the overhead, which is confirmed by the literature [25].

We conclude that ExMap is already well suited to provide fast memory allocations and I/O access to secondary storage.

D. SYSTEM-CALL INTERFACES

In this section, we compare the cost of invoking ExMap through its different interfaces. While the invocation overhead diminishes for batched operations in comparison to the actual workload, the invocation latency for short operations becomes an important factor if we want to use ExMap for memory allocation. To focus on the invocation, we allocate memory with our sequential/local member benchmark. Further, we introduce the possibility to disable batched allocations, while the free step remains batched.

In Fig. 8, we show the average operation time per allocated page, which also means that the combined system-call latency for batched allocations is 512 times higher. Please note that there is no page-fault (PF) result for batched allocations. First, we see that the effect of the invocation path indeed diminishes for batched allocations and that the per-page time increases by a factor of 5.6 from one thread to 128 threads.

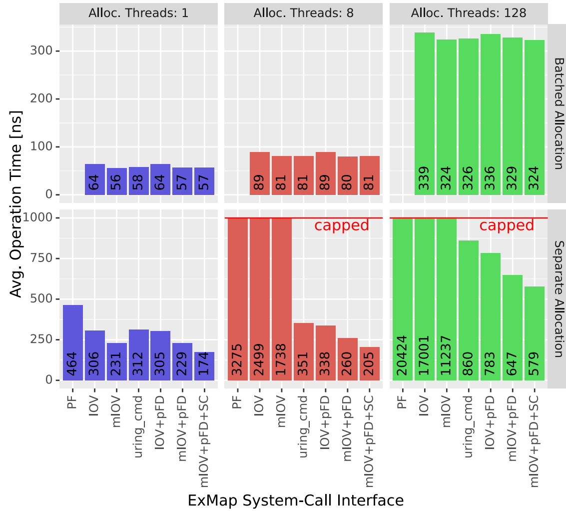


FIGURE 8. **Overhead of synchronous system-call interfaces with bulk and single-page allocations.**

For the separate allocations, we see that the page-fault path (PF) starts out slow and becomes 44 times slower with 128 threads as each page fault has to first identify the VMA object, which still [16] requires read-locking the whole VAS and a VMA-tree lookup. This overhead is the cost of PF's implicity, which is not shared by the other interfaces, where the user explicitly addresses the ExMap object (and one specific control interface) with the system-call arguments.

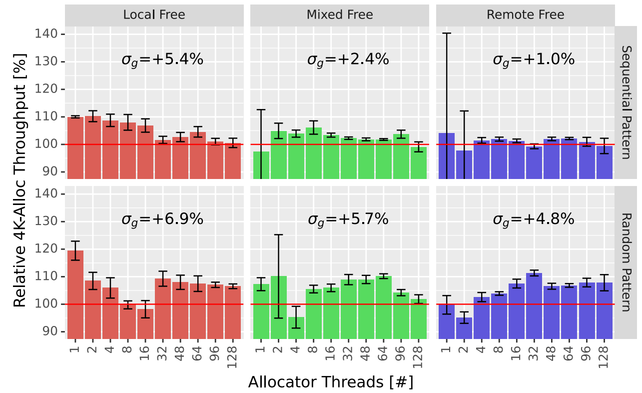
Next, we compare the traditional I/O vector and our densely-encoded permanently-mapped mIOV. Here, mIOV outperforms IOV by 25 percent with one thread and by 34 percent for 128 threads. Despite this relative benefit of mIOV, the absolute latency is unacceptable high (see *capped* bars in Fig. 8) due to the `fget` problem (see Sec. III-A).

For the next three bars, we compare three variants: (1) For uring_cmd, we use `io_uring` cmd with a registered buffer for parameter transfer and a registered file descriptor, which avoids the refcounting problem. (2) For IOV+pFD, we use ExMap's `struct iovec` allocation interface but with a private file descriptor (pFD). (3) For mIOV+pFD, we combine the mIOV with the private descriptor.

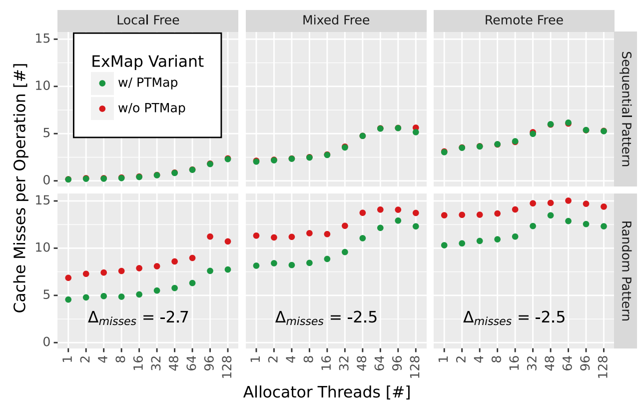
We see that registered descriptors indeed avoid the `fget/fput` false sharing problem for a high number of threads. However, `io_uring` also has some overhead for enabling asynchronous operations, which are not necessary for synchronous VAS modifications. While our private descriptors also avoid false sharing, they further are less expensive than `io_uring` and the relative advantage of mIOV remains for one thread (−25 %) and for 128 threads (−17 %).

As a last variant, we extended ExMap with its own system call (+SC), which requires manipulating the system-call table from the ExMap kernel module and using an unused system-call number as there is no API for dynamically registering a system-call number. With this variant, we explore what could be achieved with a "proper" Linux system call. With a custom system-call, we avoid the `ioctl` security checks and achieve an additional 24 percent speedup over mIOV+pFD for single-threaded operations.

Overall, we reduced the system-call overheads for ExMap by 43 percent for one thread and by 97 percent for 128 threads. Further, we learn that the asynchronous `io_uring` cmd interface is a valid choice for kernel extensions, even if they only want to provide synchronous operations.



(a) Relative 4K Allocation Performance



(b) LL-Cache Misses

FIGURE 9. **Impact of exported page tables.**

E. EXPORTED PAGE TABLES

In this section, we quantify the impact of exporting the last level of the page-table tree to the user space. This has an influence on both the space overhead and the performance of ExMap. Again, we focus on the allocation case to bring ExMap to its limit without being limited by a real SSD.

For the PtMap variant, we remove page-state tracking (see Fig. 4) from the benchmarks and replace it by the *read-only* PtMap. Instead of locking the page from user space, the pattern only queries the PtMap if a page is already *present* and, if not, we optimistically forward the page number to the allocate stage. If ExMap signals that the allocation succeeded, the thread knows that it is now the owner of that page and is responsible for freeing it again.

For the space overhead, PtMap removes the necessity for a page-state array just to know which page is present.

Since the size of the page-state array reflects the size of the underlying storage medium, it can become quite large and, as we have characterized (see Sec. IV-B), induce cache misses for random workloads. For 1 TiB of surface area (2^{28} 4K pages), a page-state array requires 32 MiB for 1-bit page state page and 256 MiB with one-byte page state. In comparison, on x86, PtMap exposes 2^{19} page tables to the user, which is the inherent cost of having a 1 TiB VMA and sharing it with the ExMap surface. Still, for its VMA, PtMap requires 1 024 *additional* page tables and one additional page directory (total: ∼4 MiB). Therefore, PtMap has an even lower space overhead than an externally-stored bit vector.

While using less memory, enabling PtMap also has a performance impact compared to the baseline ExMap performance (see Fig. 9a, y-axis starts at 90%). Thereby, the impact is twofold: (1) Each alloc–free pair requires two memory modifications less, which impacts all workloads regardless if they purely work from cache or not. (2) Querying the page state induces no additional cache miss as the subsequent memory access requires a page-table walk which also relies on the cache for accessing the page-table tree.

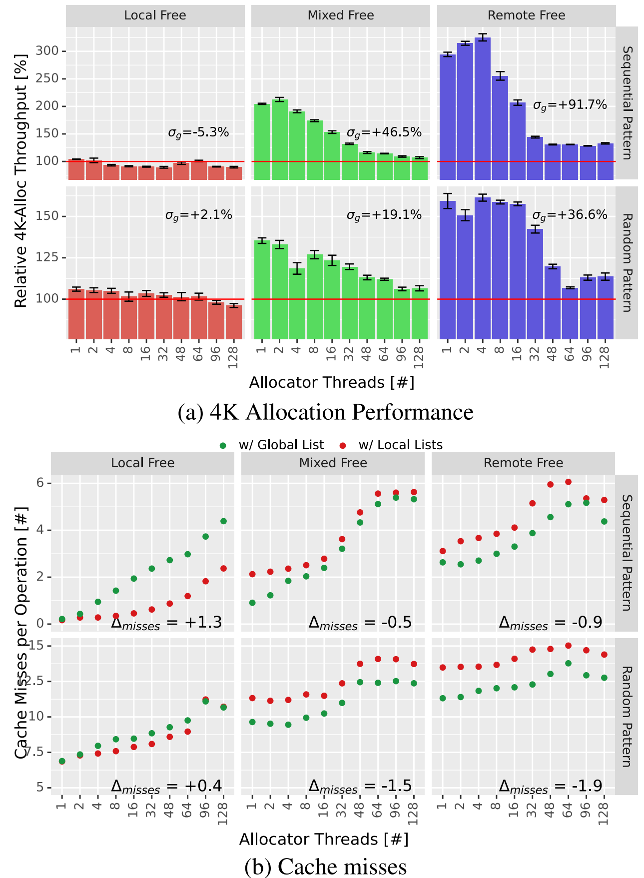
In Fig. 9a, we see both positive effects. For the sequential pattern, only the lesser modifications come into play and we see a geometric-mean improvement of up to 5.4 percent. For the random pattern, we see a significant decrease in the per-operation cache misses in Fig. 9b. The observed savings of around 2 cache misses per operation match our theoretical expectations from our benchmark characterization (see Fig. 9b). Due to the benchmark complexity, we can neither confirm nor deny whether the prefetching has the same influence here as without ExMap. Nevertheless, if we compare the sequential and the random pattern, we see a clear benefit from these saved cache misses, especially with remote free.

Another concern without user-space page-state tracking are surface conflicts that occur if two threads want to claim the same non-present page. With the page-state array, the first thread locks the page, before it asks ExMap for the page, and all operations will succeed if there is enough memory. With PtMap, both threads perform the system call and ExMap synchronizes the conflict at the PTE with the CAS operation. However, at least for our random benchmark, we observed that less than 0.009 % of all operations in this tight-loop benchmark fail due to a conflict.

Overall, we see that the PtMap enhancement is able to reduce ExMap's space overhead and increase its performance at the cost of provoking a tiny fraction of surface conflicts.

F. MEMORY-POOL MANAGEMENT

In this section, we investigate whether the *global bundle list (GBL)* outperforms ExMap's current stealing mechanism. For this, we perform the allocation throughput test. However, as the page stealing is also dependent on the number of free pages in ExMap's memory pool, we also investigate whether the pool's fill level has a significant performance impact.



(a) 4K Allocation Performance

(b) Cache misses

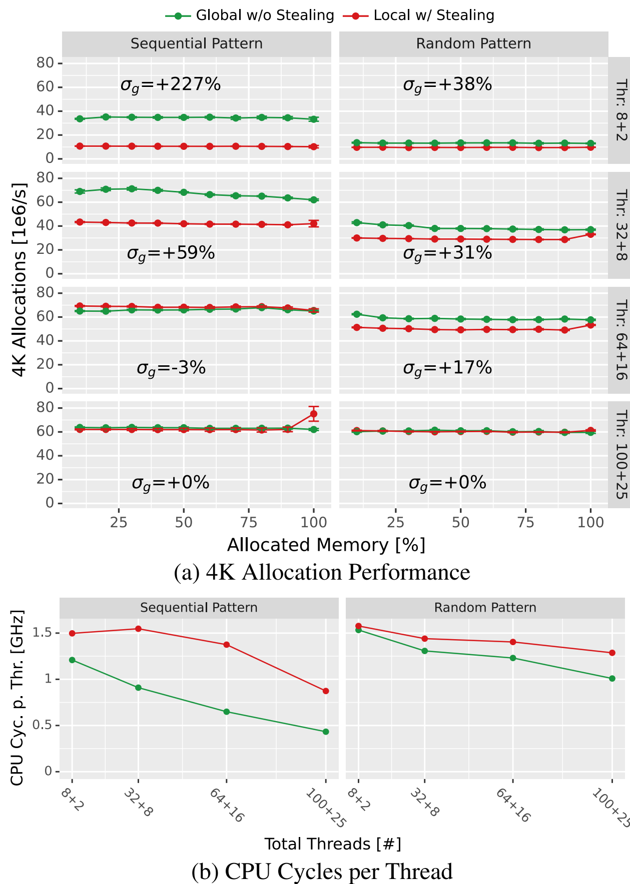
FIGURE 10. Global bundle list *without* memory pressure.

In Fig. 10a, we compare the existing ExMap with inter-interface stealing against our GBL variant. In this experiment, as for all previous, the queues can hold 50 percent of the memory pool, whereby the 50 percent are available for allocations. Further, the number of allocator and free threads remains balanced for now. The first observation is that the GBL outperforms the stealing variant in all but the local/sequential case, where it is 5.3 percent slower (geometric mean). Here, no memory has to migrate between interfaces as allocations and frees balance out at every interface, whereby the GBL provokes more cache misses per operation (see Fig. 10b). However, by switching to the random pattern, which induces a higher cache pressure, the GBL already outperforms baseline ExMap, although it has a slightly higher per-operation cache-miss rate.

When some or all pages are given to a free worker, the GBL is up to 92 percent better than the existing strategy (sequential/remote). Here, it also shows relatively stable improvements on the average number of per-operation cache misses. Nevertheless, with higher degrees of concurrency other effects start to dominate the throughput.

1) MEMORY PRESSURE

In the next step, we investigate the influence of memory pressure on both memory-pool designs. For this, we only



(a) 4K Allocation Performance

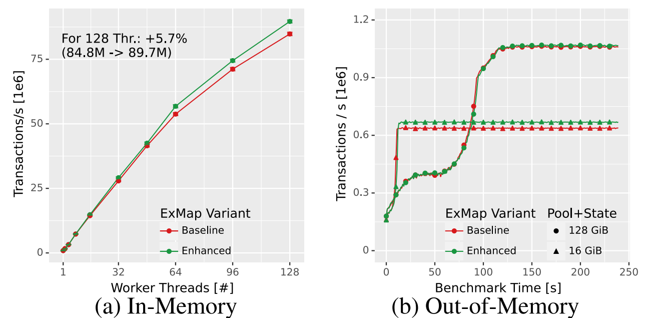
(b) CPU Cycles per Thread

FIGURE 11. **Global bundle list *with* memory pressure.**

perform tests with the remote-free strategy, where the allocation and free interfaces never match. Furthermore, we modify our benchmark as follows: First, we reduce the number of free threads to 25 percent of the number of allocator threads, so we can be certain that memory is freed slower than it is being allocated; therefore, the queues fill up and the allocator threads will regularly block when pushing their page batch to the queue. Second, we reduce the number of queues to one global queue, whereby we avoid queues running empty at the cost of having more contention. Third, we vary the size of the queue between 10 percent and 100 percent of the memory-pool size. Due to these changes, the absolute throughput of the setup is not comparable to the other performed benchmarks, but it allows for a relative comparison between stealing and the GBL.

In Fig. 11a, we see that both methods are not particularly sensitive to the number of pages in the pool. Though, for stealing, we see an unexpected speedup at 100 percent of allocated memory in cases where the method should need to collect memory from many interfaces more often. Our explanation is that, if nearly no memory is available at the interfaces, ExMap uses an optimized steal path and just grabs the whole free list instead of shaving off only a few pages, which involves linear iteration. Besides this anomaly, we see that the GBL is either on par with stealing or outperforms it. In the case of the sequential pattern, we even see a speedup of 227 percent (geometric mean) with a small number of threads. But even for the more challenging random pattern, the GBL is up to 38 percent faster.

Further, we see that GBL's relative advantage shrinks with more allocator threads. However, for this benchmark scenario, the maximal throughput seems to be at around 60 M IOP/s, which is already reached at 64+16 threads (sequential). With more threads, the same workload is only distributed over more cores, which have more than enough time to complete the operation. We confirm this by looking at the average number of executed CPU cycles per thread (see Fig. 11b) over all allocation ratios. As we have fewer software threads than hardware threads, this number can be compared to the CPU frequency (2.0 GHz, Boost: 3.7 GHz). We see that each benchmark executes for fewer and fewer CPU cycles each second, which confirms our stated hypothesis. Furthermore, the GBL requires fewer CPU cycles for an equal or higher number of allocations per second, making it even more attractive for actual workloads. For example, for 100 allocators, the GBL requires 50 percent (sequential) and 22 percent (random) fewer cycles while reaching the same allocation throughput.



(a) In-Memory (b) Out-of-Memory

FIGURE 12. **Random lookups on B+Tree. In-Memory: 100M entries ≈14 GiB. Out-of-Memory≈981 GiB.**

G. END-TO-END COMPARISON

Lastly, we perform an end-to-end comparison between the original ExMap and our enhanced variant with the improved system-call interface, the PtMap, and the GBL. For this, we use the published [7] VmCache buffer manager, which builds upon ExMap, and comes with a B+tree that supports variable-sized keys and optimistic locking. The buffer manager uses `pread()` to read from disk, so at most one I/O request is in flight per thread. To make the comparison fair, we insert the same number of entries into the B+tree but shrink the baseline's memory-pool size by the space required for the page-state array (2 GiB DRAM per 1 TiB disk).

For the in-memory case (see Fig. 12a), we insert 100 million entries (8 byte key, 120 byte payload) into the B+tree, which results in ≈14 GiB of data. With a varying number of threads, we perform random lookups. As we use a pool size of 128 GiB, all lookups are serviced from

memory, whereby the surface is not modified during the benchmark and only the PtMap-optimized in-memory path has an impact. With our enhancements, we can increase the in-memory lookup performance by a further 5.7 percent for 128 threads.

For the out-of-memory case, we (see Fig. 12b) insert 7.1 billion entries, which results in ≈981 GiB of data. Again, we perform random lookups on the tree. As the pool size becomes important with this out-of-memory scenario, we perform the experiment with two memory sizes: 16 GiB and 128 GiB. As noted, we shrink the original variants pool by 2 GiB to account for the page-state array. In Fig. 12a, we see that for the large pool sizes our enhancements have a small (+0.63 % for $t \in [140, 240]$) impact for the steady state. However, for the smaller pool, where 2 GiB increases the number of available pages significantly, the enhanced variant is 4.9 percent better.

Overall, we see that our improvements, especially PtMap, improve both the in-memory and the out-of-memory path of the already highly scalable VmCache buffer manager.

V. DISCUSSION

A. SECURITY CONSIDERATIONS OF PtMap

PtMap makes PTEs - typically a kernel-only data structure - visible, unveiling physical addresses. As this may raise security concerns at first glance, we will now discuss the potential implications. Importantly, the bare knowledge of a physical address is not inherently valuable, but significance only arises from the usage of the frame.

PtMap only unveils PTEs pointing to frames in our ExMap memory pool, as we align the start and end of the surface VMA to huge-page boundaries. Moreover, since ExMap instances are not shareable among processes, we solely leak frame addresses utilized for a process's VAS to the process itself. Hence, we do not leak physical addresses deployed for other (either kernel or user-space) mappings.

Nevertheless, one could argue that this information could indirectly shed light on the kernel's address layout, potentially jeopardizing KASLR [26]. And although PtMap reveals which pages *are not* utilized for the kernel mapping, modern operating systems shift and randomize their mapping within the kernel VAS. As a result, having knowledge of a subset of physical addresses does not provide clues about the location of kernel code or data.

However, to counter safety concerns for critical environments, we could restrict access to the PtMap feature to privileged processes through a process capability. This would still leave ExMap functioning as normal for unprivileged processes, while making the benefits of PtMap available for special applications with high performance demands.

In addition, it is worth noting that exposing physical addresses to user space is commonplace for kernel-bypass techniques. For instance, the *physical-address mode* of DPDK [27] uses an identity mapping for its memory buffers to enable the user-space setup of DMA requests. In fact, this mode is considered the default [28] for DPDK, making it functional even on systems without an IOMMU.

B. FURTHER PtMap USE CASES

Nevertheless, exposing the physical addresses could also benefit kernel-bypass interfaces, like SPDK [3]: With SPDK, the application directly issues NVMe commands into the memory-mapped command queues, which requires an address for the DMA transfer. To make this secure, SPDK creates a DPDK [27] memory pool which contains pinned (unswappable) memory that is also available in the device's I/O-VAS via an I/O-MMU mapping. In a partial kernel-bypass mode, the user could use ExMap to allocate and free memory on the surface, extract the physical address from the PTE, and hand it to the device, which accesses the memory through an I/O-VAS where all physical addresses from the ExMap memory pool are visible. Thereby, one could combine the benefits of bypassing the Linux block layer (see Fig. 6c) and having a file-mapped I/O VMA.

C. GLOBAL-BUNDLE LIST

We replace the original distributed frame pool, which requires stealing, with the GBL. While this has a positive impact on performance (see Sec. IV-F), it also reduces the implementation complexity and provides a more predictable frame-allocation overhead. For the GBL to work, we require that the bundle-stack page is mapped in the kernel's address space, which we achieve through the kernel's linear mapping of the physical memory. Nevertheless, for platforms or operating systems without such a mapping, we can still use the GBL if we map the bundle-stack page while it is used as an interface-local bundle.

D. GENERALIZABILITY

Like PtMap, our other enhancements can also have useful applications outside of ExMap: The GBL is a scalable container for the allocation of unordered and reusable resources within a shared-memory system. As an allocation has to consult exactly two locations (local bundle, GBL head), it is more predictable than stealing from a distributed thread-local pool but comes at the cost of a bounded resource stranding at the local bundles. Further, it is cheap to estimate the current memory usage from the GBL fill level without iterating over many resource containers. These properties make the GBL well suited for integration into other OS primitives.

Applications that induce many synchronous system calls that take a file descriptor as an argument will run into the `fput`/`fget` problem. With our proxy file descriptors, we provide a solution and make the idea of registered thread-local descriptors widely available without forcing applications to rewrite their logic around the asynchronous `io_uring` interface. Further, having a thread-local memory region for arguments could reduce the copy overhead for other low-latency system calls.

VI. RELATED WORK

A. SYSTEM-CALL INTERFACE

FlexSC [29] uses pinned system-call pages for issuing commands to system-call threads, which however by design only provides asynchronous system calls. Similarly, as it is inspired by FlexSC, `io_uring` [18] uses memory-mapped command queues, registered buffers, and file descriptors, but suffers from the overhead of asynchronous commands.

B. FRAME MANAGEMENT

Song et al. [30] also use process-local and CPU-pinned free lists to recycle pages, However, they do not use page bundles to reduce contention at the GBL. *Bonsai* [31] and *RadixVM* [16] provide lock-free page faults by using a balanced RCU tree and radix tree for identifying the faulty VMA. However, allocating a frame is then up to the specific VMA backend, which will usually use the system-wide frame allocator. *FastMap* [17] uses spin-locked per-core free lists and also performs page stealing. While *Aquila* [32] extends this to a two-level (core, NUMA) schema, where list-of-lists are managed at the NUMA level, they still stick with the stealing approach but on the NUMA level. In contrast, the GBL removes the cost and unpredictability of stealing and reduces cache pressure as one cache line on the stack page contains multiple pages.

C. EXPORTED PAGE TABLES

With `/proc/pid/pagemap`, Linux already supports accessing the page tables of a process. However, it only provides access via `read()` as it has to translate the information to a platform-agnostic PTE format. With Aegis [33], [34], the user managed their own page tables, which gives precise "in core" information, but relies on software-filled TLBs, which are not available anymore. Before hardware support for extended page tables (EPT), Xen [35] exported a guest VM's page table to the VM for read only access and trapped write accesses to it. Similar to Aegis, *Dune* [36] uses EPT to let the user-process manipulate its own page tables. While the mentioned methods also export page-table structures, they were proposed to allow user-level control over the VM and require a fundamentally different kernel design. In contrast, PtMap integrates easily with Linux and has the goal to only provide fast in core information.

VII. CONCLUSION

In this paper, we explored the performance characteristics of ExMap with a family of synthetic benchmarks and an end-to-end comparison of random B+tree lookups. While ExMap is able to saturate a single high-throughput SSD, the Linux block layer prohibits ExMap from achieving full throughput from a software RAID-0 array with eight disks. With *memory-mapped I/O vectors* and *private file descriptors*, we decrease the latency of single-page allocations by up to 97 percent. By managing free frames with the *global bundle list*, we increase the throughput for unbatched allocations by up to 227 percent. Also, by exposing last-level page tables with PtMap, we reduce not only the space overhead for user-space page-state tracking, but also speed up random allocations by up to 6.9 percent.

ACKNOWLEDGMENT

The authors thank their reviewers for their valuable and constructive feedback. They used ChatGPT version 4 to improve the grammar and readability of this article; the resulting output was manually verified and reflects the authors' genuine insights.

REFERENCES

[1] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim, "HBM (High Bandwidth Memory) DRAM technology and architecture," in *Proc. IEEE Int. Memory Workshop (IMW)*, May 2017, pp. 1–4, doi: 10.1109/IMW.2017.7939084.

[2] D. Jung, *2.5-Inch PCIe SSD Specification*, document PM1733, Version 1.1, Samsung, Suwon-si, South Korea, 2019.

[3] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "SPDK: A development kit to build high performance storage applications," in *Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci. (CloudCom)*, Dec. 2017, pp. 154–161.

[4] T. Neumann and M. J. Freitag, "Umbra: A disk-based system with in-memory performance," in *Proc. CIDR*, 2020, pp. 1–7.

[5] A. Crotty, V. Leis, and A. Pavlo, "Are you sure you want to use MMAP in your database management system?" in *Proc. Conf. Innov. Data Syst. Res. (CIDR)*, 2022, pp. 1–7.

[6] D. Durner, V. Leis, and T. Neumann, "On the impact of memory allocation on high-performance query processing," in *Proc. 15th Int. Workshop Data Manage. New Hardw.*, Jul. 2019, pp. 1–9.

[7] V. Leis, A. Alhomssi, T. Ziegler, Y. Loeck, and C. Dietrich, "Virtual-memory assisted buffer management," in *Proc. ACM SIGMOD/PODS Int. Conf. Manage. Data (SIGMOD)*. New York, NY, USA: ACM, Jun. 2023, pp. 1–14, doi: 10.1145/3588687.

[8] G. Graefe, H. Volos, H. Kimura, H. A. Kuno, J. Tucek, M. Lillibridge, and A. C. Veitch, "In-memory performance for big data," *Proc. VLDB Endowment*, vol. 8, no. 1, pp. 37–48, 2014.

[9] H. V. Jagadish, D. F. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan, "Dalí: A high performance main memory storage manager," in *Proc. VLDB*, 1994, pp. 48–59.

[10] J. M. Hellerstein, M. Stonebraker, and J. Hamilton, "Architecture of a database system," *Found. Trends Databases*, vol. 1, no. 2, pp. 141–259, 2007.

[11] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann, "LeanStore: In-memory data management beyond main memory," in *Proc. IEEE 34th Int. Conf. Data Eng. (ICDE)*, Apr. 2018, pp. 185–196.

[12] W. Effelsberg and T. Haerder, "Principles of database buffer management," *ACM Trans. Database Syst.*, vol. 9, no. 4, pp. 560–595, Dec. 1984.

[13] N. Amit, "Optimizing the TLB shootdown algorithm with page access tracking," in *Proc. USENIX ATC*, 2017, pp. 27–39.

[14] M. K. Kumar, S. Maass, S. Kashyap, J. Veselý, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "LATR: Lazy translation coherence," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 651–664, Nov. 2018.

[15] N. Amit, A. Tai, and M. Wei, "Don't shoot down TLB shootdowns!" in *Proc. 15th Eur. Conf. Comput. Syst.*, Apr. 2020, pp. 1–14.

[16] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "RadixVM: Scalable address spaces for multithreaded applications," in *Proc. 8th ACM Eur. Conf. Comput. Syst.* New York, NY, USA: Association for Computing Machinery, Apr. 2013, pp. 211–224, doi: 10.1145/2465351.2465373.

[17] A. Papagiannis, G. Xanthakis, G. Saloustros, M. Marazakis, and A. Bilas, "Optimizing memory-mapped I/O for fast storage devices," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2020, pp. 813–827.

[18] J. Axboe. (2019). *Efficient IO with io_uring*. [Online]. Available: https://kernel.dk/iouring.pdf

[19] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *Proc. 8th Symp. Operating Syst. Design Implement. (OSDI)*. Berkeley, CA, USA: USENIX Association, 2008, pp. 43–57.
[20] M. Cekleov and M. Dubois, "Virtual-address caches. Part 1: Problems and solutions in uniprocessors," *IEEE Micro*, vol. 17, no. 5, pp. 64–71, Sep./Oct. 1997, doi: 10.1109/40.621215.
[21] M. Dimakopoulou, S. Eranian, N. Koziris, and N. Bambos, "Reliable and efficient performance monitoring in Linux," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2016, pp. 396–408, doi: 10.1109/SC.2016.33.
[22] J. Axboe. *Deadline IO Scheduler Tunables*. Accessed: Dec. 4, 2023. [Online]. Available: https://www.kernel.org/doc/Documentation/block/deadline-iosched.txt
[23] J. Axboe. (2022). *Flexible I/O Tester*. [Online]. Available: https://github.com/axboe/fio
[24] B. Gregg, "The flame graph," *Commun. ACM*, vol. 59, no. 6, pp. 48–57, May 2016.
[25] S. Koh, J. Jang, C. Lee, M. Kwon, J. Zhang, and M. Jung, "Faster than flash: An in-depth study of system challenges for emerging ultra-low latency SSDs," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Nov. 2019, pp. 216–227, doi: 10.1109/IISWC47752.2019.9042009.
[26] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proc. 11th ACM Conf. Comput. Commun. Secur.*, Oct. 2004, pp. 298–307.
[27] Intel. (2023). *Data Plane Development Kit (DPDK)*. [Online]. Available: https://www.dpdk.org/
[28] A. Burakov. (Jun. 2019). *Memory in DPDK Part 2: Deep Dive Into IOVA*. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/memory-in-dpdk-part-2-deep-dive-into-iova.html
[29] L. Soares and M. Stumm, "FlexSC: Flexible system call scheduling with exception-less system calls," in *Proc. OSDI*, vol. 10, 2010, pp. 33–46.
[30] N. Y. Song, Y. Son, H. Han, and H. Y. Yeom, "Efficient memory-mapped I/O on fast storage device," *ACM Trans. Storage*, vol. 12, no. 4, pp. 1–27, Aug. 2016.
[31] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Scalable address spaces using RCU balanced trees," in *Proc. 17th Int. Conf. Architectural Support Program. Lang. Operating Syst.* New York, NY, USA: Association for Computing Machinery, Mar. 2012, pp. 199–210, doi: 10.1145/2150976.2150998.
[32] A. Papagiannis, M. Marazakis, and A. Bilas, "Memory-mapped I/O on steroids," in *Proc. 16th Eur. Conf. Comput. Syst.* New York, NY, USA: Association for Computing Machinery, Apr. 2021, pp. 277–293, doi: 10.1145/3447786.3456242.
[33] D. R. Engler, S. K. Gupta, and M. F. Kaashoek, "AVM: Application-level virtual memory," in *Proc. 5th Workshop Hot Topics Operating Syst. (HotOS-V)*, 1995, pp. 72–77.
[34] D. R. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel: An operating system architecture for application-level resource management," in *Proc. 15th ACM Symp. Operating Syst. Princ. (SOSP)*. New York, NY, USA: ACM Press, 1995, pp. 251–266, doi: 10.1145/224056.224076.
[35] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. 19th ACM Symp. Operating Syst. Princ.* New York, NY, USA: ACM Press, Oct. 2003, pp. 164–177, doi: 10.1145/945445.945462.
[36] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged CPU features," in *Proc. 10th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2012, pp. 335–348.

YANNICK LOECK received the B.S. and M.S. degrees in electrical engineering from Leibniz University Hannover, Germany, in 2018 and 2021, respectively. He is currently pursuing the Ph.D. degree in computer science with the Hamburg University of Technology, Germany.

His research interests include memory management and provisioning in operating systems, as well as disruptive memory technologies.

CHRISTIAN DIETRICH received the B.S. and M.S. degrees in computer science from Friedrich–Alexander University Erlangen–Nuremberg, Germany, in 2012 and 2014, respectively, and the Ph.D. degree in computer science from Leibniz University Hannover, Germany, in 2019.

Since 2021, he has been an Assistant Professor with the School of Electrical Engineering, Computer Science and Mathematics, Hamburg University of Technology, Germany. His research interests include high-performance operating-system primitives, non-volatile memory, and semi-dynamic variability.

Prof. Dietrich was a recipient of the Science Price Hannover 2020 (Leibniz University Society Hannover e.V.). He received the Dissertation Award from the GI/ITG Fachgruppe Operating Systems 2021. He is an Associate Editor of *Leibniz Transactions on Embedded Systems*.