HugeMap: Optimizing Memory-Mapped I/O with Huge Pages for Fast Storage

Ioannis Malliotakis, Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas

Institute of Computer Science (ICS), Foundation for Research and Technology – Hellas (FORTH), Heraklion, Greece
{jmal,apapag,maraz,bilas}@ics.forth.gr

Abstract. Memory-mapped I/O (mmio) is emerging as a viable alternative for accessing directly-attached fast storage devices compared to explicit I/O with system calls. Mmio removes the need for costly lookups in the DRAM I/O cache for cache hits, as they are handled in hardware via the virtual memory mechanism. In this work we present HugeMap, a custom mmio path in the Linux kernel that uses huge pages for file-backed mappings to accelerate applications with sequential I/O access patterns or large I/O operations. HugeMap uses huge pages to reduce CPU processing in the kernel I/O path compared to regular mmap. We explore the benefits and trade-offs of huge pages in HugeMap using microbenchmarks, IOR, and an in-house persistent key-value store designed for mmio. Our experiments show up to 3.7× higher throughput and up to 4.76× lower system time, compared to regular page configurations.

Keywords: Memory-mapped I/O · mmap · Huge pages · Fast storage

1 Introduction

Today, the common approach to access persistent data (e.g., a file or device) is to use read/write system calls. To improve I/O latency and throughput, the Linux kernel uses DRAM caching in the form of a page cache. In addition, applications often employ user-space DRAM caches that allow for custom policies and reduce system call overhead, which can further increase performance.

Another approach to access persistent data is to use memory-mapped I/O (mmio) i.e., Linux mmap. The user can map a file into the process virtual address space. If the requested page is not mapped, a page fault occurs, the kernel allocates a free page, reads the data from the device or file, and updates the page table. The user can access data using regular load/store instructions. Under memory pressure, the kernel evicts I/O pages to reclaim DRAM space.

Also with the Department of Computer Science, University of Crete, Greece.
Mmio provides several benefits compared to explicit I/O, e.g., read/write system calls. With explicit I/O, every I/O operation, including hits, requires an explicit cache lookup which introduces CPU overhead, even if the cache is maintained in user space [6]. On the other hand, mmio removes cache lookups for hits. In this case, if a page is cached, a valid translation in the page table exists and the cache lookup is handled in hardware by the Memory Management Unit (MMU). Additionally, mmio allows for application-specific optimizations. It can remove the serialization and deserialization in the common path: Applications access data using load/store instructions and this facilitates using the same format for both in-memory and on-device data. Furthermore, it eliminates memory copies between user and kernel space as opposed to system calls. Given these advantages, there are several attempts to use mmio in data intensive applications [10].

A disadvantage of mmio is that it produces small-sized I/Os to the underlying storage devices. This stems from the fact that the default page size is 4 KB. This significantly reduces I/O performance for sequential accesses. In addition, although this is not a significant issue for random accesses [10], being able to issue large I/Os can still improve performance by reducing overheads in the kernel I/O path. To generate large write I/Os, Linux tries to merge smaller I/Os for consecutive device blocks into larger requests, which incurs CPU overhead. In particular, many HPC applications are designed to issue large I/Os. In these cases, the 4 KB page granularity introduces overhead due to the increased number of page faults.

In this paper, we present HugeMap, a custom mmio path in the Linux kernel that uses huge pages for file-backed mappings. Our goal is to generate large I/Os where possible and accelerate sequential accesses. Today, x86_64 processors support both 2 MB and 1 GB huge page sizes. In the rest of this paper we consider only huge pages of size 2 MB, as this page size is enough to achieve peak device throughput. Currently, Linux supports huge pages only for anonymous mappings (i.e., not backed by a file or device), which are mainly used for memory allocation (i.e., malloc) [2]. We extend the mmio path with a preallocated buffer of huge pages used only for file-backed mappings, including optimizations for prefetching and fault-around operations. Additionally, we remove merges and complex asynchronous write-backs in the write path. Thus, the CPU processing needed in the common path is reduced, as we show in our evaluation. Our optimizations in the write path also benefit the msync system call that synchronizes the memory with the device for file-backed memory mappings. Finally, using huge pages reduces TLB pressure. Huge pages require a single TLB entry for a 2 MB contiguous memory area, whereas the same memory area would require 512 TLB entries of 4 KB small pages. This is an important effect as TLB size does not increase proportionally with DRAM size.

We evaluate HugeMap using microbenchmarks, IOR [8], and Kreon [10], an in-house persistent key-value store. Our results show that HugeMap achieves up to 54% and 3.7× higher throughput for sequential reads and writes respectively, relative to the corresponding regular page configurations.
2 Background

**Linux Huge Pages:** The Linux kernel offers two distinct alternatives for huge pages: transparent huge pages (THP) and HugeTLB pages. THP [14] uses sets of 512 sequential, 4 KB pages which are asynchronously and aggressively promoted to a 2 MB huge page by the `khugepaged` kernel daemon. Under conditions of memory pressure, huge pages are demoted back to sets of 512, 4 KB pages and memory compaction is performed. The behaviour of `khugepaged` with respect to page promotion, demotion, and scanning of base page sets, as well as the activation or deactivation of THP are controlled via the sysfs pseudo-filesystem. THP is currently only supported for anonymous mappings and tmpfs/shmem, therefore, we cannot directly use this mechanism over an underlying device/file.

Upon a page fault in the Linux kernel, the function `handle_mm_fault` is central to the fault handling process. In the Linux kernel, a module can register a custom page fault handler for a specific Virtual Memory Area (VMA). This is done through the virtual memory operations struct (`vm_operations_struct`), a member of the `vm_area_struct` (virtual memory area struct). The `vm_operations_struct` contains a function pointer field for handling huge page faults. The Linux kernel calls this handler for huge page faults only in the case where THP is enabled. To bypass these kernel restrictions, THP is activated with the intent of setting the necessary flags for our huge page fault handler to be called; however, huge pages are explicitly allocated by `HugeMap` through the `alloc_pages` function for the DRAM page pool. Therefore, the huge pages allocated by `HugeMap` are not actually handled by `khugepaged`, which only scans sets of 4 KB pages; all huge page operations, such as evictions, swaps, and write-backs are handled by `HugeMap`.

HugeTLB pages [13] are anonymous huge pages residing in kernel space, in a separate pool. These pages must be statically allocated by the user via the HugeTLB pseudo-filesystem (hugetlbfs). A user with the necessary privileges can then mount these pages on a pseudo-filesystem of type `hugetlbfs` with the `mount` command and use them with `mmap`. Due to their static and predefined nature, applications need to be purposely optimized to efficiently use HugeTLB pages. Additionally, as HugeTLB pages only use anonymous mappings and cannot be swapped out under memory pressure, they are not well suited for data intensive applications over fast storage devices.

**FastMap:** FastMap [11] is an optimized `mmio` path in the Linux kernel that provides a scalable manner to access fast storage devices in multi-core servers. In order to achieve scalable performance, FastMap uses three main optimizations: (1) It maintains clean and dirty pages in separate per-core data-structures, (2) it uses full reverse mappings to keep track of which page tables map a specific page, and (3) it provides a dedicated DRAM cache for increased scalability and to reduce interference with the Linux page cache. FastMap supports only 4 KB pages. `HugeMap` extends FastMap to use 2 MB pages and also provides specific optimizations for huge pages.
### 3 Design

In this section we outline the design of *HugeMap*, which creates a custom *mmio* path in the Linux kernel from the user down to the device. We implement *HugeMap* as a dynamically loaded kernel module, operating transparently to the user, either directly over the device, or over an underlying filesystem. This is determined at load-time, through the `ioctl` interface.

*HugeMap* uses a pre-allocated and configurable in size pool of huge DRAM pages (2 MB). To avoid interfering with the Linux kernel page cache, *HugeMap* maintains a separate memory pool. To provide an efficient page allocation scheme, we use per-core free lists. When the local free list is empty, we steal an empty page from another core. We always return a page to the free-list from which we originally allocated it. Similar to the Linux kernel page cache, we use a radix tree to keep track of pages that are cached by *HugeMap*. This radix tree contains both clean and dirty pages, provides lock-free lookups by using Read Copy Update (RCU), and requires locking for updates.

Furthermore, we keep dirty pages in a separate red-black tree, sorted by page device offset. To separate metadata for dirty pages, we require that a page fault occurs for every write in a read-only page. To achieve this, regardless of the user assigned *mmap* flags, we create read-only mappings in the page table. In the case of a write, an additional page fault occurs that marks the page dirty and inserts it into the red-black tree. In the case where the first access to a page is a write, for optimization purposes we combine these steps and avoid the additional page fault. Keeping the dirty pages sorted in a separate data structure allows us to have efficient *msync* and write-back mechanisms.

When there are no free pages for allocation to serve a page fault, an eviction occurs. In this case we free only clean pages. We have to update the page table, invalidate the associated TLB entries and free the page. It is also necessary to periodically write-back dirty pages to the underlying file/device, so as to have clean pages available to serve page faults. In *HugeMap*, the write-back converts a dirty page to a clean page by writing the page data into the backing device. Furthermore, it has to update the page table to mark the entry as read-only and invalidate the associated TLB entry. A set of threads asynchronously perform these tasks in order to always have clean pages available for eviction. The write-back process is triggered once the amount of dirty pages exceeds 75% of all pages in the page pool. In both eviction and write-back operations the page selection policy is LRU. For this purpose, we keep separate queues for clean and dirty pages. This approach also reduces contention by allowing evictions and write-backs to proceed concurrently.

In the case of sequential accesses, the utilization of huge pages reduces the number of page faults by $2\text{MB}/4\text{KB} = 512 \times$. Furthermore, it also reduces the complexity of write-back and *msync* operations. In these cases, for 4 KB pages we are forced to perform I/O merging to generate large I/Os from sequentially indexed pages, which is a CPU intensive process. In the case of 2 MB pages there is no need to produce even larger I/Os as (i) it is enough to achieve peak device throughput and (ii) even the Linux kernel does not support issuing larger I/Os.
for fast storage devices. Huge pages also reduce the overhead of TLB shootdowns. During a TLB shootdown, a core sends a TLB invalidation to all other cores by using inter processor interrupts (IPIs), which cause large overheads and limit scalability [1]. As these invalidations occur in larger granularity, their overhead is less pronounced.

Finally, we provide an efficient msync operation. In this case we need to write all dirty pages to the underlying device. With huge pages, this operation always produces 2 MB requests without any merges. Furthermore, we need to move all dirty pages from the dirty queue to the clean queue, update page table entries marking each entry as not writable and dirty, and then invalidate the associated TLB entries. To retrieve all dirty pages, we iterate over the red-black tree. We remove pages from the red-black tree in a batched manner, rather than one at a time, at the end of the msync operation.

**Huge Page Fault Handling Path:** In this section we present the full path of the huge-page fault handling mechanism in HugeMap. In the event of a page fault, HugeMap first searches the radix tree to check if the requested page already exists in the DRAM cache. If we find the page in the radix tree, then the requested page contains valid data and no I/O is required. Furthermore, it also resides in the appropriate clean or dirty queue. In this scenario, we only update the page table entry with the correct mapping.

If the requested page is not present in the radix tree we try to allocate a free page from the free lists. If we find a free page, we add it to the clean queue, issue an I/O to the underlying device, add the page to the radix tree and finally update the page table, without a TLB invalidation. If we cannot find a free page, we evict a configurable amount of clean pages (we use 16 in our evaluation). Eviction first removes the page from the clean queue, the radix tree, and the page table; then, after a TLB invalidation it inserts the page in the free list.

Write-back threads asynchronously write dirty pages to the backing store, remove pages from the red-black tree, move them from the dirty queue to the clean queue, update the page table and finally, perform TLB invalidations for the associated pages.

Finally, if a write request is issued to a read-only page, the page is already present in the page table as read-only. Thus, the page is moved from the clean to the dirty queue and is inserted to the red-black tree. Last, the associated page table entry is updated and the corresponding mapping is marked as writable, without requiring a TLB invalidation. Figure 1 showcases the algorithm followed by the huge-page fault handler.

**Implementation:** HugeMap is built over FastMap [11] and provides a user interface for accessing both block devices and file systems. In both cases we use our custom mmap function. All other requests, including read/write calls, are forwarded to the underlying device or file system. Implementation-wise, the utilization of huge pages over devices/files presents a few caveats besides the need to enable THP as explained in Sect. 2. Most notably, one may allocate 2 MB of contiguous memory by calling alloc_pages with the proper order argument, however, the kernel still allocates a set of 512, 4 KB pages. To treat these pages
as a single entity (huge page), it is necessary to refer to the page set only through
the first page in the 2 MB range. The corresponding *struct page* is treated as a
“representative” for the huge page and is used by the various data structures of
*HugeMap*, as well as the I/O requests issued to the underlying device. Furthermore, the *GFP_COMP* flag is used on page allocation to mark the page set as
compound [5], similarly to the existing kernel huge page mechanisms.

We also note that *HugeMap* uses the *pgoff* field of the *vm_fault* struct for
several purposes. *vm_fault* is the struct used to pass information to a page fault
handler regarding a page fault. The *pgoff* field describes the offset of the page
fault from the beginning of the device/file, expressed in pages. *HugeMap* inser-
tions and lookups to the page radix tree use it as a key, while the round-robin
selection of a per-core clean and dirty page list also relies on it. Additionally, we
require the page offset to issue I/O requests to the proper base device sector/file
offset. The *pgoff* field is, however, expressed in multiples of 4 KB pages both
for regular and huge page faults, meaning that all page faults are 4 KB page
aligned. Thus, one must adjust it in order to issue 2 MB aligned I/O requests
to the underlying device/file. In our case, this was accomplished by applying a
proper bitmask to the *pgoff* field, so that the offset points to the beginning of
a 2 MB page set. Presently, our implementation supports defining the page size
at compile time via a preprocessor macro. Thus, our module can also work with
regular (4 KB) pages as described in Sect. 2. We use this setup for our evaluation.

4 Methodology and Evaluation

Our testbed consists of a dual-socket server that is equipped with two Intel(R)
Xeon(R) CPU E5-2630v3 CPUs running at 2.4 GHz, each with 8 physical cores
and 16 hyper-threads for a total of 32 hyper-threads. The storage device of the
server is a PCIe-attached Intel Optane SSD DC P4800X series with 375 GB capacity. This server is equipped with 256 GB of DDR4 DRAM at 2400 MHz and run CentOS v7.3, with Linux kernel 4.14.72. We disable swapping and CPU frequency scaling to reduce variability in our measurements. In all cases we run the experiments three times and report averages.

First, we use a custom microbenchmark that maps a block device and issues I/O accesses (memcpy) using multiple threads. It supports both sequential and random accesses. As HugeMap only supports regular (4 KB) and huge (2 MB) pages, we only evaluate sequential accesses with a single and multiple threads.

Furthermore, we use the Kreon [10] key-value store for our evaluation. Kreon is a persistent key-value store that trades random device I/O patterns for lower CPU consumption. This is possible as modern storage devices (e.g. SSDs and NVMe) provide high I/O throughput even with small I/Os under high concurrency. Kreon relies on mmap to interact with storage. It uses a log for allocations, Copy-On-Write (CoW) for persistence and provides scalable insert and lookup operations by using fine-grained locking. The use of the log produces a sequential write access pattern for insert (or update) only workloads; this type of pattern is well-suited to benefit from the use of huge pages.

How does HugeMap perform with sequential I/O patterns? Figure 2 shows how mmap performs with an increasing number of threads under different configurations. We compare HugeMap (both with regular and huge pages) with Linux mmap. For Linux we use the madvise system call to inform the kernel of the expected I/O pattern. We use both the MADV_RANDOM and MADV_SEQUENTIAL options. The latter does aggressive read-ahead that can potentially improve sequential performance.

For HugeMap with regular pages and Linux with MADV_RANDOM we see similar throughput. As we increase the number of threads we observe higher throughput. This happens because of the higher queue depth in the device. Both Linux and HugeMap achieve peak throughput with 16 or more threads.

HugeMap with huge pages and Linux with MADV_SEQUENTIAL, with 2 or more threads both achieve peak device throughput. With 32 threads, huge pages result in about 12% higher throughput compared to the configurations with regular pages. This shows that high device queue depth is not enough to achieve peak device throughput. Finally, with 1 thread HugeMap achieves 54% higher throughput compared to Linux with MADV_SEQUENTIAL. Although in Linux the aggressive read-ahead also results in large reads from the device, it requires a page fault per 4 KB, rather than 2 MB in the case of HugeMap.

Table 1 shows device performance for all the previously discussed cases, using 32 threads. With huge pages (or aggressive read-ahead) we achieve the peak device throughput of about 2.5 GB/s. Thus, high concurrency to the device is not enough to achieve peak throughput. In all cases we have 100% device utilization. Finally, HugeMap for a sequential pattern requires a page fault per 2 MB, instead of 4 KB (i.e., 512× fewer page faults). This results in lower CPU overheads and larger concurrency to the device (i.e., higher queue depth).
With \textit{mmap} a write to a page results in a read-modify-write operation, as the kernel does not know if a page contains useful data. On sequential writes over \textit{HugeMap} using 32 threads, huge pages result in a $3.7\times$ higher throughput compared to regular pages, with the former achieving throughput equal to 1526 MB/s and the latter 412 MB/s.

Specifically, for the write-only microbenchmark (Table 1), due to the read-modify-write operation we observe both reads and writes. \textit{HugeMap} with huge pages always generates 2 MB I/O requests, with minimal CPU processing. Thus, \textit{HugeMap} with huge pages achieves 4.5% and 3.44% higher read and write throughput respectively compared to regular pages. This stems from the fact that with regular pages, all read requests are 4 KB and merging is performed for the write requests. The average request size is 11.38 sectors. In that case the greater number of page faults and the CPU-hungry I/O merging does not allow the microbenchmark to reach 100% device utilization. Finally, \textit{HugeMap} with huge pages provides $74.5\times$ higher I/O queue size.

![Fig. 2. Throughput scalability for a read-only microbenchmark](image1)

![Fig. 3. Execution time breakdown for an insert-only workload with \textit{Kreon}](image2)

|                    | Read (MB/s) | Write (MB/s) | \text{avg rq (sectors)} | \text{avg qz} | util (%) |
|--------------------|-------------|--------------|-------------------------|--------------|----------|
| \text{HugeMap 4 KB reads} | 2261        | –            | 8                       | 32.2         | 115      |
| \text{HugeMap 2 MB reads}  | 2527        | –            | 256                     | 351          | 100      |
| \text{Linux (MADV RANDOM) reads} | 2277   | –            | 8                       | 28.7         | 114      |
| \text{Linux (MADV SEQUENTIAL) reads} | 2543   | –            | 256                     | 31.2         | 101      |
| \text{HugeMap 4 KB writes}  | 273         | 286          | 10.4                    | 2.3          | 32.5     |
| \text{HugeMap 2 MB writes}  | 1229        | 985          | 256                     | 170          | 100      |

\text{avg rq}: Average size of requests issued to device  
\text{avg qz}: Average queue length of requests issued to device  
\text{util}: Percentage of CPU time in which I/O requests were issued to device

How does \textit{HugeMap} impact CPU consumption? In this section we examine how much \textit{HugeMap} affects CPU consumption for both read-only and write-only microbenchmarks. Figure 4a shows the execution time breakdown for the
Fig. 4. Microbenchmark execution time breakdown.

Fig. 5. CPU time breakdown for IOR.

Table 2. Device performance for IOR Checkpoint (left) and Restore (right).

|                | xput (MB/s) | avg_rq (sectors) | avg_qz | util (%) |
|----------------|-------------|-----------------|--------|----------|
| Read/write     | 1963        | 413             | 700    | 14.5     |
| mmap (THP)     | 1940        | 253             | 529    | 4.1      |
| mmap (no THP)  | 1928        | 257             | 479    | 3.9      |
| HugeMap        | 1892        | 429             | 30     | 73.2     |

read-only microbenchmark. In both HugeMap with regular pages and Linux with MADV_RANDOM system time is about 10%. In the case of Linux with MADV_SEQUENTIAL, system time is 2.3% and in the case of HugeMap with huge pages system time is 0.14%. In all cases the majority of execution time is iowait time, which means that the device is the bottleneck. Reducing system time leaves more CPU processing capacity for the user application.

Figure 4b shows that for the write-only microbenchmark the use of huge pages in HugeMap reduces the percentage of system time from 94.37% to 20.39%. As we use a microbenchmark in this case the user time in both cases is very low (below 1%). The remainder of system time in the case of huge pages goes to iowait and idle time. This means that an even faster storage device will result in even better performance.
Does **HugeMap** improve checkpoint and restore in HPC? We use the IOR [8] parallel I/O benchmark to evaluate the advantages and drawbacks of **HugeMap** regarding CPU usage and device utilization compared to read/write system calls (without **THP**) and Linux **mmap** (with and without **THP**). We ran experiments on our testbed using an NVMe Optane device. We use two different scenarios in this case. The first scenario is **Checkpoint**, in which 8 processes concurrently write to the NVMe device, for an aggregate write size of 160 GB, or 20 GB per process. This scenario aims to emulate saving the program state in a large scale parallel system. The second scenario is **Restore**, where 8 processes concurrently read the previously written files from the NVMe device, in order to emulate the system restoring itself to a previously saved state. We use 50 GB of main memory in both benchmarks and this includes all page mappings. We report the average over 5 repetitions of each scenario. Before each benchmark the kernel page cache is completely cleared so that no previously cached page is available.

Table 2 showcases these results. The left columns correspond to the **Checkpoint** scenario. In this case we observe that read/write system calls achieve the higher throughput, with Linux **mmap** following closely in performance. **HugeMap** achieves 3.75% lower throughput compared to read/write system calls which is close to the maximum achieved throughput. The columns on the right contain the results for the **Restore** scenario. In this case, we can see that **mmap** achieves 43% lower performance compared to read/write system calls. On the other hand, for **Restore**, **HugeMap** shows the highest performance out of all configurations, with a 44% and 0.6% improvement compared to Linux **mmap** and read/write system calls respectively.

Figure 5a demonstrates the execution time breakdown for the **Checkpoint** scenario. Here we see that **HugeMap** requires slightly greater system time compared to read/write system calls (11% compared to 6%) to achieve almost the same performance, while also enjoying the benefits of **mmio**. Figure 5b shows the CPU breakdown for the **Restore** scenario. In this case, **HugeMap** achieves 9× lower system time compared to read/write system calls and 18.5× compared to **mmap**. Combined with the improvement in throughput, it becomes apparent that **HugeMap** provides the best behaviour for this scenario.

Does **HugeMap** benefit key-value stores? Finally, we use **Kreon** in order to provide a more realistic evaluation under a more complex workload. We use **YCSB** [4] benchmark with an insert-only workload and a dataset of 10M entries (about 10 GBs of keys and values). We provide enough DRAM to ensure that data fits in memory. This experiment examines the impact of 512× fewer page faults on **YCSB** throughput and the balance of system and user CPU time.

Our benchmarks indicate that with huge pages **Kreon** achieves 89.5% higher throughput in terms of ops/s, 1.99 Mops/s compared to 1.05 Mops/s with regular pages. Figure 3 shows that huge pages reduce system time from 30.8% to 6.5%. On the other hand, user time increases from 52.7% to 88%. In both cases iowait percentage is the same and the idle percentage is low compared to user and
system pages, we reduce system time and we leave more CPU processing capacity for the user application, i.e. Kreon and YCSB.

5 Related Work

We briefly review prior work related to huge page management and mmio for fast storage devices. Ingens [7] and Hawkeye [9] provide several optimizations for anonymous page mappings in Linux, mainly in the path of promotions and demotions of regular to huge pages and vice-versa. They modify the THP mechanism in Linux (Sect. 2) and currently operate only on anonymous mappings. HugeMap on the other hand focuses on file/device backed huge page mappings, which are significantly different from anonymous mappings. DI-MMAP [15] is a custom mmio path in the Linux kernel that tries to optimize it for HPC applications. It uses a dedicated DRAM cache and also provides a FIFO-based eviction policy that is optimized for this type of applications. The authors in [12] optimize the mmio path in the Linux kernel. The main improvements are in the case of free page allocation and Vectored I/O that optimize write operations, thus showing that fast storage can be used to efficiently extend the available DRAM size. FastMap [11] shows that the Linux mmio path suffers from scalability limitations with more than 8 threads and provides an evaluation for both storage applications and for extension of DRAM over fast storage devices. In all cases, regular (4 KB) pages are used. HugeMap uses FastMap and demonstrates the benefits of huge pages in storage applications. The authors in [3] propose the use of read-ahead mechanisms to provide increased throughput for sequential access patterns. We also provide optimizations for sequential access patterns, however, we use huge pages for this purpose. Our approach not only generates larger I/O requests but also reduces the number of page faults and TLB misses.

6 Conclusions

In this paper we present HugeMap, a custom memory-mapped I/O path inside the Linux kernel that uses only huge pages for I/O operations. Our approach achieves peak device throughput for sequential I/O patterns. HugeMap reduces system time by (1) reducing the number of page faults and TLB invalidations to one per huge page, and (2) reducing the need to perform I/O merging during page write-back. We evaluate HugeMap with microbenchmarks, IOR, and the Kreon persistent key-value store. Our results show that HugeMap improves I/O throughput by up to up to 3.7× and reduces system time by up to 4.76×. Although we do not explore this further, we believe that HugeMap can eventually support a hybrid approach, combining regular and huge pages dynamically.

Acknowledgements. We thankfully acknowledge the support of the European Commission through the H2020 project EVOLVE (GA825061) and the support of the Hellenic General Secretariat for Research and Technology through the MIA-RTDI Research-Create-Innovate project SentiTour (T1EDK-02857). Anastasios Papagiannis is also supported by the Facebook Graduate Fellowship.
References

1. Amit, N.: Optimizing the TLB shootdown algorithm with page access tracking. In: 2017 USENIX Annual Technical Conference (USENIX ATC 2017), Santa Clara, CA. USENIX Association, July 2017
2. Brown, N.: Transparent huge pages in the page cache (2016). https://lwn.net/Articles/680690/
3. Choi, J., Kim, J., Han, H.: Efficient memory mapped file I/O for in-memory file systems. In: 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 2017), Santa Clara, CA. USENIX Association (2017)
4. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing. SoCC 2010. Association for Computing Machinery, New York (2010)
5. Corbet, J.: An introduction to compound pages. https://lwn.net/Articles/619514/
6. Harizopoulos, S., Abadi, D.J., Madden, S., Stonebraker, M.: OLTP through the looking glass, and what we found there. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. SIGMOD 2008, Association for Computing Machinery, New York (2008)
7. Kwon, Y., Yu, H., Peter, S., Rossbach, C.J., Witchel, E.: Coordinated and efficient huge page management with Ingens. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016), Savannah, GA. USENIX Association, November 2016
8. Loewe, W., McLarty, T., Morrone, C.: The IOR Benchmark (2012). https://github.com/hpc/ior
9. Panwar, A., Bansal, S., Gopinath, K.: HawkEye: efficient fine-grained OS support for huge pages. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS 2019. Association for Computing Machinery, New York (2019)
10. Papagiannis, A., Saloustros, G., González-Férez, P., Bilas, A.: An efficient memory-mapped key-value store for flash storage. In: Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018. Association for Computing Machinery, New York (2018)
11. Papagiannis, A., Xanthakis, G., Saloustros, G., Marazakis, M., Bilas, A.: Optimizing memory-mapped I/O for fast storage devices. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20), Boston, MA. USENIX Association (2020)
12. Song, N.Y., Son, Y., Han, H., Yeom, H.Y.: Efficient memory-mapped I/O on fast storage device. ACM Trans. Storage 12(4) 1–27 (2016)
13. The Linux Kernel Documentation: HugeTLB Pages. https://www.kernel.org/doc/html/latest/admin-guide/mm/hugetlbpage.html
14. The Linux Kernel Documentation: Transparent Huge Page Support. https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html
15. Van Essen, B., Hsieh, H., Ames, S., Pearce, R., Gokhale, M.: DI-MMAP—a scalable memory-map runtime for out-of-core data-intensive applications. Cluster Comput. 18(1), 15–28 (2015)