Kernel level support for transparent use of huge-pages in memory mapped I/O

Ioannis Malliotakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Angelos Bilas*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

Kernel level support for transparent use of huge-pages in memory mapped I/O

Thesis submitted by
Ioannis Malliotakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: Ioannis Malliotakis

Committee approvals:

Angelos Bilas
Professor, Thesis Supervisor

Kostas Magoutis
Associate Professor, Committee Member

Polyvios Pratikakis
Assistant Professor, Committee Member

Departmental approval:

Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, November 2022

Kernel level support for transparent use of huge-pages in memory mapped I/O

Abstract

Memory-mapped I/O (*mmio*) allows applications to transparently access data in storage devices via the page-fault mechanism, using processor load/store instructions. *mmio* has the potential to (a) eliminate modifications to applications for handling and processing large datasets by merely extending their heap over fast storage devices and (b) provide attractive abstractions for an application to control its I/O path over a unified data representation. Despite these advantages, *mmio* has significant limitations that make it less attractive. In this thesis, we first discuss the current limitations of *mmio*. Then, we design *xmap*, an alternative *mmio* implementation for the Linux kernel, which addresses these limitations. The main contributions of *xmap* are support for transparent huge pages over block-based storage and asynchronous promotions. To our knowledge, *xmap* is the first system that provides this support for the Linux kernel. We evaluate *xmap* with a variety of graph processing workloads using Ligra, an in-memory graph processing framework, by transparently extending its heap over storage with no code modifications. Our results show that when processing graph datasets 6-8× larger than the available system DRAM, *xmap* outperforms Linux *mmap* by up to 3.5×, reduces total page faults by up to 265×, decreases CPU system time by up to 90%, and increases CPU user time by up to 250%.

Υποστήριξη επιπέδου πυρήνα για την διαφανή χρήση μεγάλων σελίδων σε είσοδο/έξοδο χαρτογραφημένης μνήμης

Περίληψη

Η είσοδος/έξοδος χαρτογραφημένης μνήμης (*mmio*) επιτρέπει στις εφαρμογές να προσπελάσουν με διαφανή τρόπο δεδομένα σε συσκευές αποθήκευσης μέσω του μηχανισμού σφάλματος σελίδας, χρησιμοποιώντας εντολές φόρτωσης/αποθήκευσης του επεξεργαστή. Το *mmio* έχει την προοπτική (α) να καταστήσει περιττές τις τροποποιήσεις στις εφαρμογές για την διαχείριση και την επεξεργασία μεγάλων όγκων δεδομένων επεκτείνοντας τον σωρό (heap) τους πάνω από συσκευές αποθήκευσης και (β) να παρέχει ένα ελκυστικό επίπεδο αφαίρεσης για το μονοπάτι εισόδου/εξόδου μίας εφαρμογής χρησιμοποιώντας μία ενοποιημένη αναπαράσταση για τα δεδομένα. Παρά αυτά τα πλεονεκτήματα, το *mmio* έχει σημαντικούς περιορισμούς οι οποίοι το καθιστούν λιγότερο ελκυστικό. Σε αυτήν την εργασία, αρχικά αναλύουμε τους παρόντες περιορισμούς του *mmio*. Στη συνέχεια, σχεδιάζουμε το *xmap*, μία εναλλακτική υλοποίηση του *mmio* για τον πυρήνα Linux, το οποίο αντιμετωπίζει αυτούς τους περιορισμούς. Οι κύριες συνεισφορές του *xmap* είναι διαφανής υποστήριξη για μεγάλες σελίδες πάνω από συσκευές αποθήκευσης βασισμένες σε blocks, και ασύγχρονες προαγωγές μεγάλων σελίδων. Το *xmap* είναι το πρώτο σύστημα που παρέχει αυτές τις δυνατότητες στον πυρήνα Linux. Αξιολογούμε το *xmap* με πληθώρα αλγορίθμων επεξεργασίας γράφων χρησιμοποιώντας το Ligra, ένα πλαίσιο επεξεργασίας γράφων που λειτουργεί εντός μνήμης, επεκτείνοντας διαφανώς τον σωρό του πάνω από συσκευές αποθήκευσης, χωρίς καμία τροποποίηση στον κώδικά του. Τα αποτελέσματά μας δείχνουν ότι όταν επεξεργαζόμαστε όγκους δεδομένων 6-8× μεγαλύτερους από την διαθέσιμη μνήμη του συστήματος, το *xmap* πετυχαίνει απόδοση έως και 3,5× καλύτερη από αυτή του Linux mmap, μειώνει τα συνολικά σφάλματα σελίδας έως και 265×, μειώνει τον επεξεργαστικό χρόνο συστήματος έως και 90% και αυξάνει τον επεξεργαστικό χρόνο χρήστη έως και 250%.

Ευχαριστίες

στην οικογένεια, τους φίλους μου, και όσους με στήριξαν αυτά τα 2 χρόνια

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data-intensive applications, such as graph analytics frameworks and key-value stores, are extensively used in cloud infrastructures to serve and process data [DeC+07; Cao+20; Zah+10; Cha+08]. Such applications are typically required to operate on datasets that exceed memory (DRAM) size. This requirement introduces two issues. First, not all applications are written to manipulate out-of-memory datasets because of the complexity this requires in moving data to and from devices in an efficient manner. In such cases, applications are limited to the size of the DRAM, although their datasets grow faster than the available memory in typical servers. Second, applications that manage large datasets typically perform explicit I/O system calls (*explicit I/O*) to control the application I/O path, incurring significant overheads owing to the system I/O software stack. A user-space cache, considered the state-of-the-art practice, avoids system calls for hits, however, still requires lookups for every hit resulting in significant software overhead.

Memory-mapped I/O (*mmio*) has the potential to overcome both of these issues. With *mmio* an application may map the contents of a file/device to its virtual address space, and access them with regular CPU load/store instructions. Accesses are served transparently by the OS kernel via the page cache. Essentially, this approach creates a large heap for the application that has two inherent advantages, as follows.

First, *mmio* can eliminate application redesign for handling large datasets: *mmio* presents a single abstraction of the data in memory and the storage device, without the need for intermediate data transformations (serialization and deserialization). The representation of data in each user-space application is inherently coupled with the application functionality and design. Explicit I/O system calls require altering the representation of data between memory and storage devices, e.g., to eliminate or transform pointers to memory locations. Consider for instance analytics frameworks, such as Spark [Zah+10] which cache intermediate computation results off-heap to a storage device. This function requires data serialization for transfers from memory (heap) to storage (off-heap) and deserialization for the reverse route, which are a significant source of overhead [Kol+20]. With *mmio*,

page faults and page writebacks eliminate serialization and deserialization through a unified data representation in a virtual address space spanning both memory and storage.

Second, *mmio* can entirely eliminate the cost of I/O cache hits. The buffered I/O approach requires a system call even in the case of hits and data copies from the kernel to user space. These overheads lead many systems to favor the use of a user-space cache coupled with *direct I/O*. Even user-space caches incur significant overhead for constantly performing lookup operations for hits. On the other hand, *mmio* incurs no software overhead for cache hits. With *mmio*, hits in the page cache imply an established page table entry and the virtual address translation performed by the MMU in hardware. This is semantically equivalent to a cache lookup. We note that although cache miss handling is significantly different between *mmio* and explicit I/O, both still require a transition to kernel space. Furthermore, we observe that the use of *mmio* for user-space storage caching is in essence another expression of the heap extension paradigm.

However, the current state-of-the-art practice is to avoid *mmio* because it has significant shortcomings as well. In particular, *mmio* generates small random I/Os and incurs high I/O overhead because it uses 4 KB pages, causes applications to stall waiting for synchronous I/O due to the nature of page faults, lacks control over the I/O path [Pap+16], and does not scale well with increasing core numbers [CKZ13]. We observe that the proliferation of fast storage devices introduces two new requirements: (a) To examine if applications can transparently process large datasets without requiring specialized design and modifications to explicitly manage data in memory and storage. (b) To design, implement and propose possible applications of mechanisms to provide finer control of the I/O path through *mmio*. The need for such mechanisms is becoming ever more apparent due to the ability to perform millions of I/O operations per second [LC22], which renders I/O cache management overheads more significant. *mmio* has the potential to satisfy both of these requirements by transparently extending the application heap.

In this work, we first discuss the shortcomings of *mmio*. Then, we present *xmap*, an alternative *mmio* path, implemented as a Linux kernel module. We build on *FastMap* [Pap+20a], which extends *mmio*, in three significant ways. First, we enable huge (2 MB) pages for file-backed mappings. Second, we implement asynchronous promotions, a process in which a huge-page sized region of the application virtual address space is elevated from using regular page table entries to one huge-page table entry, in order to reduce the overhead and latency seen by applications. Finally, we add support for finer grain control over the page fault path via *madvise* semantics.

We implement *xmap* in Linux and evaluate its transparent heap extension capabilities with real graph processing workloads. We also introduce the implemented mechanisms and semantics of *xmap* for future evaluation into finer-grain application application-driven control over the *mmio* I/O path. *xmap* requires no kernel

modifications. Applications using *mmio* can transparently use *xmap*, whereas *malloc*-based applications designed to operate in-memory may use *xmap* via libvmmalloc [Gro] without any further modifications. We examine Ligra [SB13], a graph processing framework designed to operate in-memory. *xmap* transparently extends the Ligra heap on fast storage devices allowing it to process larger graphs without any modifications. Our results show that Ligra is able to process graphs 5-8x larger than the memory size within 3.08× (average) the time of an ideal in-memory version (assuming adequate DRAM) without any modifications. *mmap* allows Ligra to process the same datasets, however, at 2.44× higher cost than *xmap*. Finally, our evaluation shows that our dynamic huge page promotion mechanism is particularly effective in generating large I/Os, reducing system overhead, and performing most of the related processing asynchronously to the application.

Chapter 2

mmio Background

***mmio* Semantics:** In the *mmio* paradigm, a user-space process initially issues a system call (*mmap* on POSIX systems, *MapViewOfFile* in Windows) to reserve ranges of its virtual address space. These ranges, or mappings as we will refer to them, may be associated with a *named object*, i.e., a file or device, be it byte-addressable (DRAM with an in-memory filesystem, NVM) or block-addressable (e.g., SSDs and HDDs), in which case they are called *file-backed mappings*. *Anonymous mappings* on the other hand are not associated with a named object and are used by the OS memory management subsystem to implement volatile memory allocations. Mappings are further split into *private* and *shared*, depending on whether writes to the mapping are visible to other processes. In this paper, we focus on file-backed shared mappings which are necessary for accessing block-addressable devices, and use the term "file" interchangeably for files and devices, as the semantics of *mmio* remain the same. Accesses to file-backed mappings over block devices are serviced through a kernel-space DRAM page cache. Pages are initialized in memory via *major* page faults. These fetch data from storage. Page cache hits are known as *minor* page faults.

System page sizes: Processors offer multiple page sizes. The most common default page size is 4KB. Such pages are called regular or base pages. Pages larger than regular pages, i.e. 2MB and 1GB pages, are called huge pages (commonly in Linux), superpages (commonly in FreeBSD), or large pages (Windows). We will refer to them as huge pages throughout this paper. Huge-page table entries occupy a single TLB entry, similar to regular page table entries, thus increasing TLB coverage. Huge-page table entries can reduce processing time spent in page fault handling, as page faults are served at coarser granularity. However, huge pages may lead to excessive memory fragmentation and unnecessary I/O for applications with relatively low spatial locality. Each virtual page (regular or huge) is required to correspond to an equally sized *contiguous* DRAM frame.

Chapter 3

Limitations of *mmio*

In this section, we discuss our insights for the limitations of *mmio* and we present the high level design of *xmap*. Today, the mainstream approach to perform I/O is to use system calls (explicit I/O) for moving data between memory and the devices. System calls can be either buffered in the kernel page cache or via direct I/O coupled with a user-space cache. Compared to these approaches, *mmio* has two inherent advantages: (a) It can entirely eliminate the I/O cache hit cost and (b) it can eliminate the need for application redesign to handle persistent data. Despite these inherent advantages of *mmio*, current implementations suffer from three significant limitations. Next, we discuss each of these in detail, along with our approach for mitigating them in *xmap*.

4KB-page granularity: Currently, the *mmio* path in Linux operates only in the granularity of regular pages (4 KB) for file-backed mappings. This results in three significant overheads: (a) Handling page faults for regular pages requires significant CPU (system) time resulting in high CPU utilization. (b) Most I/Os generated are small, which results in high device utilization and low throughput, even for fast storage devices, such as NVMe SSDs. (c) With increasing dataset sizes, TLB coverage of the virtual address space decreases drastically, resulting in more TLB misses. Although the kernel can perform batched cache insertions, evictions, and writebacks for multiple pages, it does not address these issues that are inherent to the regular (4 KB) page size. Huge pages have the potential to reduce CPU overhead for page faults, to increase I/O size, and to increase TLB coverage by more than two orders of magnitute. *xmap* integrates huge pages (2 MB) in the *mmio* path entirely transparently to the application and is able to dynamically choose the page type used to serve each page fault, based on application hints or automatic policies.

Synchronous nature of page faults: The synchronous nature of page faults is an important consideration for *mmio*: Applications need to wait for faults to be handled before they can continue execution. For major page faults, this also

includes the time for performing I/O to the underlying storage. When a page fault is combined with synchronous readahead this increases I/O and page fault handling time. Asynchronous readahead can overlap I/O time with application execution, but still requires minor page faults for each regular page. To reduce the impact of synchronous page faults, *xmap* introduces *asynchronous huge page promotions.*

Page promotion refers to elevating a set of regular page table entries for contiguous virtual addresses to one huge page table entry. The corresponding physical regular pages may need to be copied to a newly allocated huge page if they are not already contiguous. Asynchronous page promotion performs the required work for promoting a set of regular pages to a huge page in the background, while the application continues executing. As soon as the asynchronous promotion completes, the application will see a single page table entry for the huge page, with appropriate permissions, incurring no additional page fault and I/O overheads. Asynchronous huge page promotion needs to deal with four issues: (a) How to synchronize between multiple racing promotions to minimize wasted processing and synchronization costs. (b) How to fetch the missing regular pages within a huge page. (c) How to maximize promotion concurrency with application execution and page faults. (d) When to perform a promotion. We discuss these issues in depth in Chapter 3.1.

Lack of control over the I/O path: Compared to *explicit I/O*, *mmio* exhibits significant lack of control for (a) page cache replacements and (b) the size of I/O operations.

Page cache replacements: All *mmio* operations need to go through DRAM pages, e.g. the Linux buffer cache. However, there are cases where an application would like to exclude certain I/O operations from polluting the I/O cache by evicting other pages. For instance, in modern key-value stores, I/Os related to compaction operations are massive but are typically performed with direct I/O to avoid the cache. Similar cases involve I/Os to write-ahead logs that are used for recovery purposes and other types of scan or append operations that touch data only once and reduce the effectiveness of LRU replacements. Although previous work has proposed replacement algorithms [MM03; BM04] that aim to address such issues, applications tend to use *direct I/O*, which ensures that all related I/O will bypass the I/O cache. *xmap* mitigates the negative performance impact of scans by using two separate pools of pages. Page faults stemming from scans can be served from a separate pool, avoiding pollution of the main page pool used for the I/O cache.

Size of I/O operations: *mmio* issues regular-page I/Os that have a significant impact on performance. *mmio* typically supports some type of explicit or implicit readahead, but this generally is either a flat policy that users can merely turn on or off or a mechanism based on short-term access pattern history. *xmap* allows controlling the size of I/Os by using explicitly regular or huge pages in two ways.

First, an application can specify *madvise* calls for a Virtual Memory Area (VMA) region. Second, an application can create multiple mappings for a file and use different madvise calls for each VMA. The kernel treats both cases in the same manner internally. If a page fault from a regular-page-hinting mapping is handled with a regular page before a page fault from a huge-page-hinting mapping occurs to an overlapping offset, an asynchronous promotion is scheduled to migrate the corresponding file contents to a huge page. In the inverse case (a regular-page-hinting mapping finding a huge page upon a page fault) the page fault is minor and the only necessary step is installing a page table entry over the appropriate huge-page subpage. Therefore, the application can control the size of I/Os explicitly, whenever this is required, for different types of operations (read, write), different portions of files, and different intervals during application execution.

3.1 Design and Implementation of *xmap*

Figure 3.1 shows the placement of *xmap* in the kernel. *xmap* may operate either over a raw device or over a filesystem. Both approaches map a device or file range to the process virtual address space and associate the resulting VMA with a set of *xmap*-implemented operations. When operating over a filesystem *xmap* uses a modified version of *wrapfs* [ZB99], a stackable filesystem interface which uses the Linux kernel *Virtual File System* API. Applications using *xmap* interact with files within wrapfs and *xmap* is responsible to issue operations from kernel space to the underlying filesystem.

We note that *mmap*, the current *mmio* mechanism in Linux, does not scale well as the number of cores increases [CKZ12; Pap+20b]. The main reason is lock contention for both the process page cache, as well as the LRU lists used by the kernel for page replacement decisions [Pap+20a]. To address scalability issues, *xmap* builds on the alternative *mmio* path first implemented by *FastMap* [Pap+20a][1]. *xmap* extends *FastMap* without increasing synchronization overheads, with NUMA-aware page allocations, support for explicit *buffered I/O* and *direct I/O*, and an arbitrary number of reverse mappings. Then, *xmap* adds extensive support for file-backed huge pages, asynchronous promotions, and improved control over the I/O path. Next, we provide an overview of the main components and structures in *xmap* and discuss our design and implementation.

3.1.1 Overview

Figure 3.2 shows the main structures of *xmap*: (1) **per-file data** (PFD), (2) **reverse mapping information** (RMI), and (3) **page pool** (PP). These structures serve as (1) the file associated page cache, (2) the VMA page reverse mapping information, and (3) the structures comprising the page pool common for all memory

[1] https://github.com/CARV-ICS-FORTH/FastMap

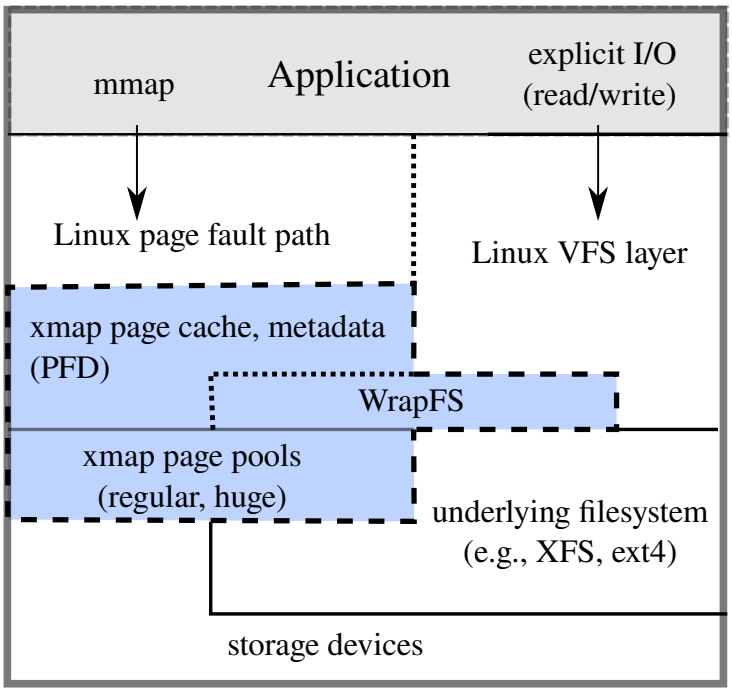

Figure 3.1: Placement of *xmap* components within the Linux kernel. *xmap* component boundaries denoted by dashed lines. Dotted lines denote component interoperability.

mappings serviced by *FastMap*. All structures in *FastMap* (and thus *xmap*) use per-core instances to reduce contention and improve scalability.

Per-file Data: Each underlying file is linked to per-file data (PFD) that include three types of information: (a) A per-core radix tree for clean and dirty pages that serves as the page cache for the mapped file. (b) A per-core red-black tree containing only dirty pages, which decouples the page writeback path from the page fault cache lookup path to reduce contention. (c) A per-core reservation list for promotions.

Reverse Mapping Information: A reverse mapping for a physical page indicates where the page is mapped within a VMA via a page table entry. With file-backed mappings, a physical page used for a mapped file will contain a reverse mapping for each VMA that points to it. Reverse mappings are added to a page in the page fault path and removed when the corresponding VMA is unmapped.

As Linux needs to accommodate use cases where pages are shared by a large number of VMAs, it implements object-based reverse mappings [Cor03; Cor04]. This approach saves memory but requires list traversals to locate all page table entries associated with a page. As a result, reverse mappings become a significant

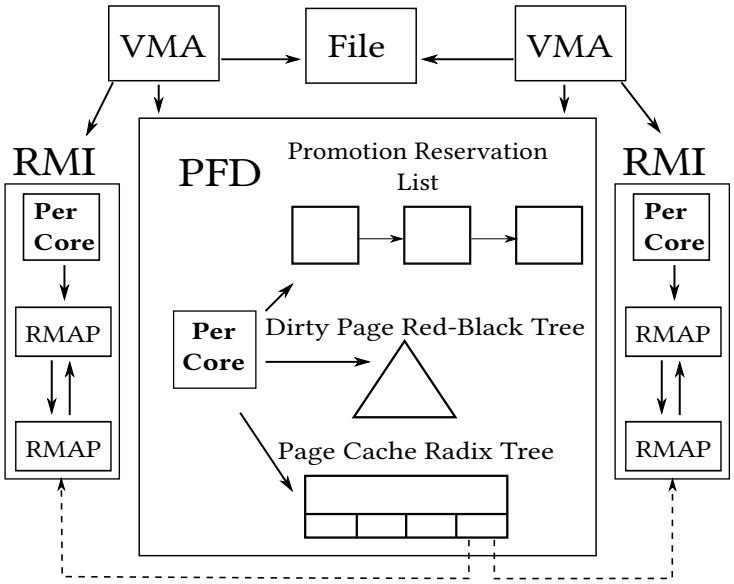

Figure 3.2: Overview of the main *xmap* components.

source of synchronization overhead.

Since *xmap* is designed around file-backed mappings for I/O purposes, the expected degree of page sharing is low, in contrast to e.g., a page belonging to code for a shared library. For this reason, *xmap* uses full reverse mappings, stored in a linked list per page. Each full reverse mapping contains a virtual address used for a page table walk to retrieve the associated page table entry. *xmap* organizes reverse mapping information (RMI) as a per-core linked list of all page reverse mappings associated with a VMA. An RMI is created when the VMA is opened. RMI (and PFD) are stored in the private data field of the kernel's VMA structure.

Similar to Linux *mmap*, *xmap* needs to support splitting of VMAs. VMA splitting occurs when unmapping part of a larger VMA with a partial *munmap* system call or via an *madvise* call which provides a hint for part of a larger VMA. When splitting a VMA, all reverse mappings of the split VMA associated with the virtual address range being split need to be transferred to the new VMA created by the split. *xmap* implements VMA splitting by transferring part of the full reverse mappings from the RMI of the starting VMA, to the new VMA created from the split. However, when the kernel calls the VMA split handler to notify *xmap* about a VMA split, the new VMA has not been created yet. Therefore, at this point, *xmap* cannot associate the mappings from the previous RMI structure with the new VMA. *xmap* circumvents this by deferring the actual splitting of reverse mappings until the new VMA is opened (after the creation process is complete). *xmap* detects if a VMA open operation is the result of a VMA split by performing a lookup on the process VMA tree with the address passed to the open call. If a different VMA (than the one being opened) is found, then the current open is

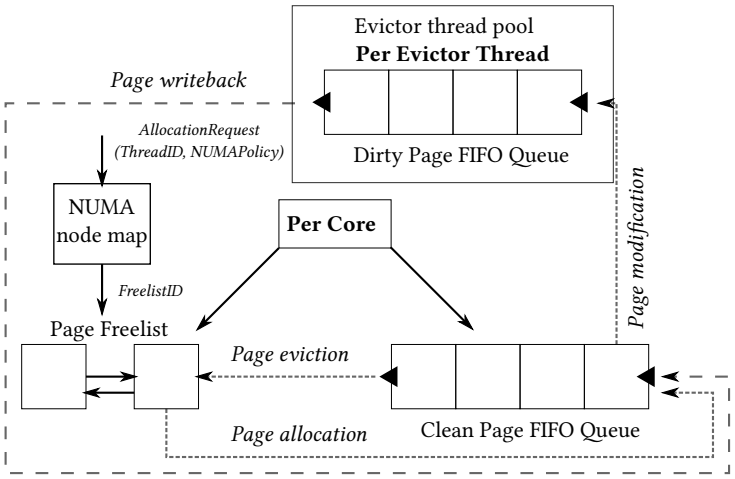

Figure 3.3: Page pool architecture in *xmap*.

the result of a split, which *xmap* then handles properly with respect to the RMI transfer.

Page Pool: *xmap* uses the page pool (PP) to organize its pages, keeping track of free, clean, and dirty pages. *xmap* splits its pages in two pools, one for regular and one for huge pages, each using separate freelists, clean and dirty queues, and writeback threads. The two pools differ in behaviour parameters, such as the amount of clean pages evicted in a batch (lower for huge pages) and batching of dirty pages in the writeback path (not required for huge pages). The aggregate memory footprint of both pools, along with the percentage of memory to be utilized by the huge page pool are passed to the module as load-time parameters. Figure 3.3 shows the high-level architecture of *xmap*'s page pool.

xmap allocates free pages at initialization time (module loading) and places them in per-core freelists. *xmap* performs this placement in a NUMA-aware manner: Pages in each core freelist are allocated from the core's corresponding NUMA node when possible. Free pages are used to satisfy page requests on major page faults. *xmap* checks the NUMA policy of the VMA to determine the order of searching the freelists. If no NUMA policy has been set, the default policy (similar to Linux) is to search freelists in ascending NUMA node distance, starting from the faulting node. To reduce contention, we search freelists within a NUMA node in random order.

Clean pages are mapped but unmodified pages. Similar to Linux, *xmap* uses an LRU approximation to evict only clean pages. They are distributed in a round robin manner to per-core FIFO queues based on the offset of the underlying file they map. This ensures that pages mapping contiguous file offsets are distributed to different queues. Consequently, page faults to contiguous offsets access different clean page queues reducing contention.

Dirty pages are pages that have been modified. They are placed in dirty queues and are candidates for writeback to storage. Writeback is performed asynchronously by a configurable amount of writeback kernel threads. To reduce contention, the page pool uses one dirty queue per writeback thread. Each writeback thread periodically checks whether the percentage of dirty pages to total pool size exceeds a threshold and writes pages from its dirty queue back to storage. The distribution of dirty pages to queues is similar to that of clean pages. However, to allow for batched writeback operations, pages mapping contiguous file offsets need to be placed in the same dirty queue. For this reason, the page offset is right-shifted by nine bits (divided by 512), which for 4KB regular pages results in a writeback thread being able to batch up to 2MB of contiguous regular pages for writeback.

3.1.2 Support for Huge Pages

Providing support for huge pages in file-backed mappings requires both interfacing to complex, existing Linux mechanisms and extending the structures inherited by *FastMap*, as well as providing new mechanisms. Interfacing with Linux requires providing two handlers. First, a "mapping" handler (for the VMA operation *get_unmapped_area*) which the kernel uses to return a starting virtual address for the *mmap* system call, aligned to a huge-page address. The default Linux handler returns regular-page aligned addresses, which is not sufficient for *xmap*.

Second, a page-fault handler (in the VMA operation structure) exclusively for huge pages. This handler is expected to completely handle all aspects of the page fault, including I/O and page table entry setup, as opposed to regular page fault handlers, which are at minimum required by the kernel only to select a page to service the fault and lock it, with the kernel then handling page table entry setup. The handler must additionally ensure for correctness that huge-page table entries are only created for eligible VMAs. These are VMAs containing a virtual address range large enough to fit the entire huge page, and aligned to the huge page size at the huge page starting offset. If these requirements are not met, then the handler may only setup regular page table entries over huge page subpages.

Figure 3.4 shows a high-level overview of *xmap*'s page fault handler. First, the huge-page selection policy function (line 1) is consulted to determine whether a huge page is to be synchronously prepared or asynchronously promoted to handle the page fault. The policy function can be any function accepting the kernel's page fault descriptor structure as its argument and returning a boolean value specifying the page type to use, true for huge, false for regular. Next, the handler performs a lookup in the page cache, and in the case of a hit proceeds to the page table entry setup step, else it allocates a page to serve the page fault (lines 4-8). If asynchronous promotions are enabled then the regular page pool is always used for synchronous page allocations on page faults, else the appropriate page pool is selected based on the policy function result. The page allocation is followed by a page cache insertion and synchronous I/O for the allocated page. Note that

```
1: UseHugePage ← HugePageSelectionPolicy()
2: pg ← CacheLookup()
3: if NoPage() then
4:     if PromotionsEnabled() then
5:         pg ← PageAllocation(Regular)
6:     else
7:         pg ← PageAllocation(UseHugePage)
8:     end if
9:     CacheInsert(pg)
10:    PageIO(pg)
11: end if
12: if PageIsHuge(pg) then
13:    if HugeEntrySupported(PgFAddr) then
14:        SetupHugePTE(PgFAddr, pg)
15:    else
16:        SetupRegularPTE(PgFAddr, Subpg(pg))
17:    end if
18: else
19:    SetupRegularPTE(PgFAddr, pg)
20:    if PromotionsEnabled() ∧ UseHugePage then
21:        SchedulePromotion()
22:    end if
23: end if
```

Figure 3.4: *xmap* page fault handling overview in the presence of huge pages and asynchronous promotions.

concurrent page faults may race to insert page cache entries for the same file offset, thus only successful insertions trigger I/O. The last step is the page table entry setup (lines 12-23). If the page allocated - or found in the cache - is huge, then the handler must check whether the faulting VMA is eligible for a huge-page table entry, and if so set up a huge-page table entry, else set up only a regular page-table entry over the appropriate huge-page subpage. If the page is instead regular, the handler sets up a regular page-table entry and if promotions are enabled and requested by the policy function, a promotion is scheduled. We also note that the huge pages allocated by *xmap* are set as compound [Cor] for three reasons: (1) For *xmap* to efficiently differentiate between the two types of pages it handles. (2) To allow for efficient Linux filesystem reads and writes through *bio_vec* structures. (3) To efficiently acquire the "representative" huge page pointer from any subpage within the huge page, as without compound pages we would have to manually set the index (page offset) field for each subpage and perform pointer arithmetic.

Next, huge pages require complex manipulation in *xmap* with respect to per-file data and reverse mappings. The Linux kernel page structure contains the index field, which describes the backing file offset mapped by the page, expressed in multiples of 4KB. This index serves as the entry key for both the page cache tree and the dirty page tree of PFD. Regular pages require only one index for the 4KB range they map. With huge pages however, the index field instead corresponds to the start of a span of page offsets mapped by the page. Therefore, special care is required as regular and huge pages coexist in the page cache and dirty page trees to avoid inconsistencies for shared regular and huge-page mappings, when accessed from different cores. Huge pages also require special attention regarding reverse mappings. Regular pages only require one reverse mapping per VMA which maps them. The reverse mappings for a huge page however, differ based on whether the VMA maps the entire huge page with a single page table entry, or subpages of the huge page with regular page table entries due to size and alignment restrictions. *xmap* always uses one reverse mapping for the huge page, which acts as "representative" for all possible subpage reverse mappings. The virtual address of the representative mapping is the lowest address within the VMA that overlaps with the huge page. When handling such mappings, e.g., when marking all corresponding PTEs clean after writeback or unmapping the page, the starting address is used for a page table walk, which then finds and handles all subpage entries.

An additional concern is TLB flushing for these entries. Similar to previous approaches [Pap+20a] that use batched TLB flushes to limit TLB shootdowns [ATW20], *xmap* performs TLB flushes at coarse granularity for huge pages. With huge pages, a reverse mapping corresponds to a virtual address range of up to 2MB. Therefore, for each huge-page reverse mapping we issue a TLB flush corresponding to the entire address range of the VMA which maps into the huge page, regardless of whether the VMA has mapped the entire huge page.

3.1.3 Support for Asynchronous Promotions

Synchronous huge-page faults prepare and fetch the huge page during page fault time resulting in high latency. To reduce latency, *xmap* introduces asynchronous promotions: It always services page faults with regular pages, schedules a promotion, and returns to the faulting application before the promotion is complete. This reduces the synchronous part of the page fault to a regular-page-sized I/O, a page table and TLB entry setup, and the scheduling of the promotion to a Linux workqueue. The asynchronous promotion ensures proper handling of page table and TLB entries. Asynchronous huge page promotions in *xmap* are designed to be completely transparent to the application. The decision to perform a huge-page promotion is taken at page fault time, through the same policy that selects a huge page to serve a page fault.

Figure 3.5 summarizes the steps involved in the asynchronous portion of the promotion. Asynchronous promotions incur significant complexity, in two respects: (a) It is important to avoid concurrent, racing promotion attempts for the same huge page from different page faults, because they result in redundant processing and I/O. (b) The asynchronous promotion mechanism needs to ensure application correctness in the possible presence of concurrent application page faults.

To handle concurrent, racing promotions *xmap* extends the PFD with per-core promotion reservation lists. Each promotion to be scheduled at page fault time first checks the appropriate PFD promotion reservation list, chosen round-robin based on the huge-page-aligned promotion offset. The list is scanned and if a reservation with the same offset is found no promotion is scheduled. Else, a reservation is placed in the reservation list (empirically we insert at the head).

Anonymous huge-page promotions in the Linux kernel (implemented by the THP mechanism, discussed in Chapter 6) obtain a write-lock on the process *mmap* semaphore and hold it for the entirety of the promotion. This is a correctness requirement, as the promotion scans process page tables to locate the regular pages involved in the promotion. The *mmap* semaphore protects against concurrent page faults, which obtain a read-lock on the semaphore, and VMA mutating operations, such as mmap/munmap, which obtain a write-lock. *xmap* only requires a lock to its page cache tree to locate pages involved in a promotion. Therefore it is possible for *xmap* to reduce the operations currently performed with the write-lock held on the semaphore, thus further increasing the mechanism's concurrency. We note however that it is currently not possible to completely eliminate the semaphore write-lock from the promotion mechanism without kernel modifications. The Linux kernel page-fault handling code does not expect a Page Middle Directory (PMD) entry to be upgraded from holding a set of PTEs to representing one huge-page table entry from under it, because it holds the semaphore read-lock while the kernel-implemented anonymous promotions require the semaphore write-lock. Therefore any implementation of huge-page promotions requires for correctness' sake a write-lock on the semaphore at least for the huge-page table entry setup step.

Finally, asynchronous promotions only operate in the context of huge pages.

```
1: ReplacedPages ← ∅
2: start ← PromotionOffset
3: end ← start + 512
4: hugePg ← PageAllocation(Huge)
5: PageIO(hugePg)
6: WriteLock(MmapSemaphore)
7: LockCache()
8: i ← start
9: repeat
10:     pg ← NextCacheEntryInRange(i, end)
11:     ReplaceCacheInRange(i,
              MIN(pg.indx, end), hugePg)
12:     if pg.indx ≥ end then
13:         break
14:     end if
15:     LockPage(pg)
16:     ReplacedPages ← ReplacedPages ∪ pg
17:     i ← pg.indx
18: until NoPage()
19: UnlockCache()
20: for all pg ∈ ReplacedPages do
21:     for all Rmap ∈ pg.ReverseMappings do
22:         RemovePTE (Rmap.VMA, Rmap.VAddr)
23:     end for
24: end for
25: for all VMA ∈ VMAsMappingRange(start, end) do
26:     FlushTLBRange(
              VMASubRange(VMA, start, end))
27: end for
28: for all pg ∈ ReplacedPages do
29:     if PageDirty(pg) then
30:         Copy (Subpg(hugePg), pg)
31:         MarkPageDirty(hugePg)
32:     end if
33:     FreePage(pg)
34: end for
35: for all VMA ∈ VMAsMappingRange(start, end) do
36:     if HugeEntrySupported(VMA, start, end) then
37:         for i ∈ SubrangeInVMA(VMA, start, end)
            do
38:             SetupRegularPTE(VMA,
                    Subpg(hugePg, i))
39:         end for
40:     else
41:         SetupHugePTE(VMA, hugePg, start)
42:     end if
43: end for
44: WriteUnlock(MmapSemaphore)
```

Figure 3.5: Pseudocode for asynchronous promotions.

Applications that require data in chunks with size between a regular and a huge page, suffer from multiple, synchronous regular page faults. Similar to (implicit) Linux read-ahead, *xmap* provides (explicit) asynchronous page prefetching in the form of an *ioctl*, which accepts a virtual address range and prefetches its regular pages asynchronously in the *xmap* cache, by issuing a separate prefetch operation per VMA within the range to the *xmap* workqueue. The ioctl returns control to the application before any actual I/O is performed. The I/O during each prefetch operation closely resembles its page-fault counterpart.

3.1.4 Promotion Policies

We implement two general-purpose promotion policies, which we then evaluate in Chapter 5. These policies are checked at page fault time, and if they determine that a huge page promotion is in order, the page fault is served synchronously with a regular page and an asynchronous promotion is scheduled. The first policy, *promote-always*, schedules asynchronous huge page promotions on each page fault, as long as the VMA in which the fault occurs satisfies size and alignment restrictions. This policy aims to take advantage of high spatial locality to use huge PTEs, reduce latency, and mask I/O cost. The second policy, *promote-seqfault*, separately tracks the faulting page offsets for each CPU core per PFD and schedules promotions when it detects two page faults by the same core to sequential page offsets, as long as mapping size and alignment restrictions hold. This policy is a simple and lightweight approach to avoiding the use of huge pages for random access patterns.

3.1.5 Hints and Directives

To allow for improved control over the I/O path, *xmap* uses the huge-page related flags of *madvise*, i.e, MADV_(NO)HUGEPAGE to allow access to a portion of the mapped file in two ways (a) spatially, with different pages sizes for different address ranges and (b) temporally, with different pages sizes during application execution. First, an application may issue two separate *mmap* system calls for the same file, and pass the MADV_HUGEPAGE flag to one of them, e.g., to the mapping passed to a scanning thread. This will create two VMAs, one that accesses the file via regular pages (VMA-regular) and one that access the file via huge pages (VMA-huge). The application can control the page size used, by directing its file operations to the appropriate VMA area. For accesses to the same file location from multiple VMAs, *xmap* performs the necessary page promotions, in the case of regular page cache hits by VMA-huge, and regular page table entry setup over huge page subpages, in the case of huge page cache hits by VMA-regular, each time an access is handled. Second, the application can use *madvise* calls with different flags during its execution for the same VMA. In this case the behavior of the VMA with respect to page size, will follow the specified flag until the next *madvise* call. This allows an application to change during its execution how a single file portion is

accessed via a VMA. With both approaches, the MADV_HUGEPAGE flag serves as an indicator to *xmap* to serve page faults from a VMA with pages from the regular page pool or the huge-page pool. Thus the huge-page pool is utilized in sequential operations, e.g., scans, without polluting the regular page pool, which serves for finer-grained caching. Although these pools can in general be designed in different forms, we find that it is adequate to have one pool for regular pages and one for huge pages, each with separate page eviction and page writeback mechanisms.

Chapter 4

Evaluation Methodology

Our experimental testbed is a dual-socket server with two Intel Xeon E5-2630 CPUs, operating at 2.4GHz. Each socket has 8 cores with 2 threads per core for a total of 32 threads. The server is equipped with 256GB of DDR4 DRAM operating at 2400MHz frequency. The system memory is split equally into two NUMA nodes, each with 16 threads. As storage in our experiments we use two Samsung MZQLB1T9HAJR-00007 NVMe SSDs in a RAID-0 setup with the XFS filesystem and an aggregate capacity of 3.6TB, and disable swap. The server runs on CentOS version 7.9 and the Linux kernel version is 4.14.123. We use Ligra with multiple workloads for our evaluation. We run each workload three times, and report average values across the runs. We present execution time statistics, execution time breakdowns (user time, system time, IO-wait time, and idle time) recorded via *mpstat*, and page fault and I/O statistics. Where appropriate, we also present *xmap*-specific statistics. I/O statistics are recorded with *iostat*.

Ligra [SB13] is a graph processing framework. It is designed to operate in-memory, by initially parsing a graph description file, the application input, and then computing a workload via maps over edges or maps over subsets of vertices. We use Ligra configured with OpenMP. As the input dataset we use a real world dataset from the Friendster social network [RA15; net], with $|V| = 65608367$, $|E| = 1806067135$. We use three variants of this graph, one with directed edges, one with undirected edges[1], and one with directed weighted edges, with the weights being randomly generated in the range $[1, log_2|V|]$. Table 4.1 summarizes the graph algorithms we run in Ligra. The size of the dataset representation in memory for all algorithms varies between 82-148 GB. These sizes allow us to explore out-of-memory configurations but also to contrast our results to "ideal", in-memory configurations. To evaluate out-of-memory configurations for Ligra, we limit the amount of memory in our servers to 18GB (16 GB for the page cache and 2 GB for the OS) via cgroups [Lin]. This creates a ratio of dataset to DRAM between 12% and 22% (16% on average) for the different graph algorithms. We

[1]Therefore, $|E_{\text{undirected}}| = 3612134270$, as each input edge is interpreted by Ligra as two edges with symmetric endpoints.

Workload Name	Short Name	Graph type dir/weight	Parameters	Required Memory
Breadth First Search	BFS	no/no	sv=52432407	~82GB
Connected Components	CC	no/no	-	~113GB
K-Core Decomposition	KCore	no/no	-	~148GB
Maximal Independent Set	MIS	no/no	-	~138GB
Graph Radii Estimation	GRAD	no/no	-	~133GB
Betweenness Centrality	BC	no/no	sv=18059434	~100GB
Single-Src Short. Path	SSSP	yes/yes	sv=17256435	~104GB
PageRank Delta	PR	yes/no	maxi=100	~85GB

Table 4.1: Ligra workloads and their parameters. *sv* and *maxi* stand for the *source vertex* and the *max iterations* parameters of Ligra. Graphs are directed/undirected and weighted/unweighted, as indicated.

use libvmmalloc [Gro] to extend the heap of Ligra over the RAID storage device. We compare the following configurations:

mmap: Default Linux *mmap* with the following sysfs parameters that remove unnecessary writebacks: dirty_background_ratio = 60, dirty_writeback_centisecs = 0, and dirty_ratio = 99. We note that *mmap* cannot use huge pages with file-backed mappings. ***xmap*-nohuge:** *xmap* with no huge pages and no page readahead, used as a baseline for base regular page performance with a scalable page *mmio* architecture. ***xmap*-huge-always:** *xmap* with synchronous huge pages but without asynchronous promotions. The page cache is configured with 98% huge pages and 2% regular pages. ***xmap*-promote-always:** *xmap* with asynchronous promotions for all page faults. The page cache is configured with 98% huge pages and 2% regular pages. ***xmap*-promote-seq:** *xmap* with asynchronous promotions and the sequential page fault policy. The page cache is configured with 50% huge pages and 50% regular pages.

We also use two in-memory configurations for calibration purposes, where Ligra runs with its heap located entirely in DRAM. We use these two setups as "ideal" configurations that would provide the best application performance, assuming the full dataset fits in memory. **in-memory-(No)THP:** In memory configuration with transparent huge pages enabled (in-memory-THP) and disabled (in-memory-NoTHP) via sysfs[2].

[2] *echo {always/never} > /sys/kernel/mm/transparent_hugepage/enabled*

Chapter 5

Experimental Results

5.0.1 Impact of *xmap* on out-of-core performance

Our evaluation of heap extension over flash-based storage using Ligra seeks to answer the following questions:

1. What is the overall performance improvement of *xmap* for out-of-core performance?
2. What is the impact of each *xmap* mechanism?

Overall performance improvement of *xmap* Figure 5.1 shows the total execution time for each Ligra workload, normalized to the "ideal" in-memory configuration with transparent huge pages. *xmap*-nohuge outperforms the Linux mmap on four of the eight evaluated workloads (BFS, CC, KCore, BC), however its configuration with no implicit page readahead is a detriment to its performance on the remaining workloads, thus its average performance penalty is 9.06×, 20% higher than Linux mmap (7.5×). Adding huge pages to *xmap* leads in overall performance improvement between 1.02× (*xmap*-promote-seq, SSSP) and 3.5× (*xmap*-promote-always, KCore) compared to mmap. The best performing *xmap*-based configuration, *xmap*-promote-always, achieves the lowest average performance penalty across all workloads among all out-of-memory configurations at 3.08×. *xmap*-promote-always is thus 2.44× and 2.94× faster on average than the regular-page-using Linux mmap and *xmap*-nohuge, respectively. Additionally, *xmap*-huge-always and *xmap*-promote-seq also universally outperform both Linux mmap and *xmap*-nohuge by 53% (*xmap*-promote-seq, 4.9× average performance penalty) to 84% (*xmap*-huge-always, 4× average performance penalty) and 85% to 123%, respectively.

Table 5.1 shows the average page faults across all Ligra workloads for each out of memory configuration. This is where the most significant performance advantage of huge pages becomes apparent. The Linux mmap is able to avoid major page faults via its aggressive readahead strategy. These page faults however still require

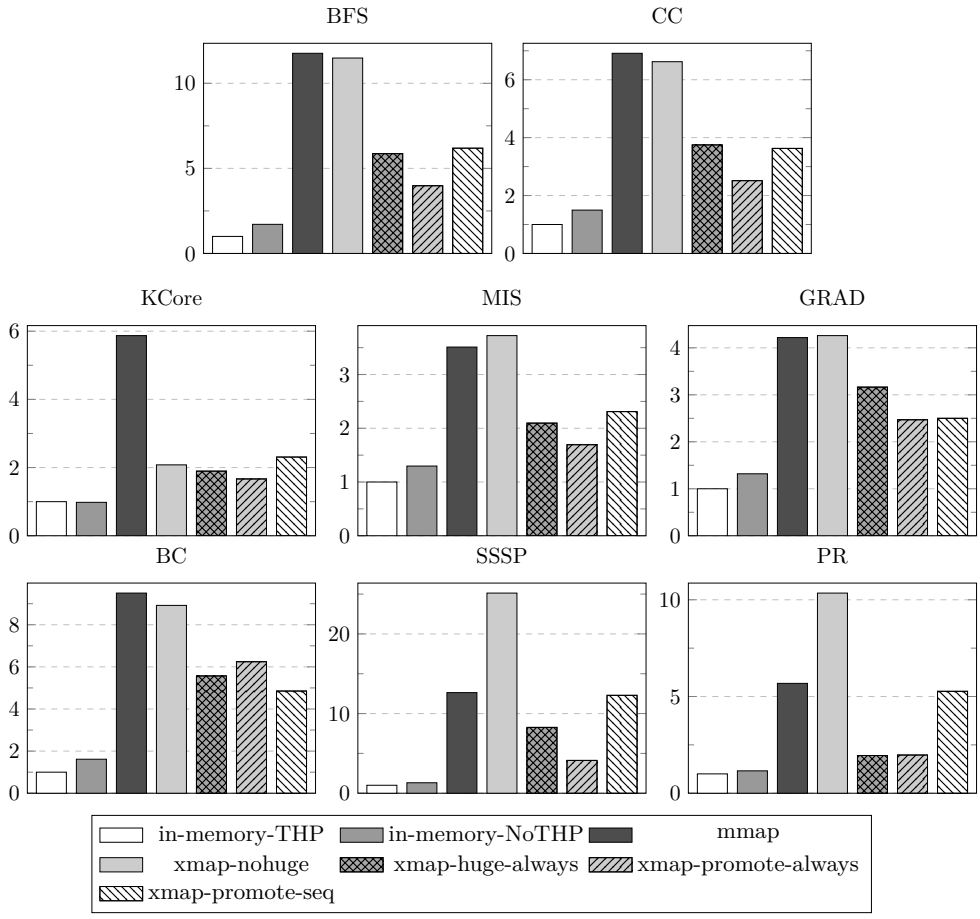

Figure 5.1: Ligra total execution time normalized to the in-memory version with transparent huge pages, as an optimal configuration that places all data in memory and uses huge pages to reduce TLB overhead.

Page Faults	Major	Minor
mmap	88,703	122,132,391
***xmap*-nohuge**	101,418,770	45,126,456
***xmap*-huge-always**	528,908	231,286
***xmap*-promote-always**	1,442,473	133,985
***xmap*-promote-seq**	30,611,253	208,088

Table 5.1: Average number of page faults across Ligra workloads for out-of-memory configurations.

switches to kernel-space for processing, which leads Ligra with the mmap configuration to spend a significant portion of its execution time in kernel-space. This is evident in Figure 5.2, which shows the total execution time breakdown for Ligra per configuration and workload, where the Linux mmap exhibits low user time (12.5% of execution on average) compared to system time (45.81% of execution on average). *xmap*-nohuge has similarly low user time percentage (12.5% average) due to major page faults, but scales better than mmap (11.3% average system time percentage) and moves the execution bottleneck to synchronous I/O stalls (34.6% average IO-Wait time). Huge-page based *xmap* configurations incur between 1.9× (*xmap*-promote-seq, BFS) and 265× (*xmap*-huge-always, KCore) less total page faults compared to mmap. On average across workloads, the amount of total page faults drops from ~4× (*xmap*-promote-seq) to ~161× (*xmap*-huge-always) compared to mmap, and from ~4.57× to ~192× compared to *xmap*-nohuge, depending on how aggressively huge pages are used. This leads to up to 3.49× higher user time percentage (*xmap*-huge-always, BFS), with the average user time percentage for the *xmap* based configurations being 1.46× (*xmap*-promote-seq, 18.24%) to 2.19× (*xmap*-promote-always, 27.41%) higher than mmap. System time percentage also drops to between 95% (*xmap*-huge-always, BFS) and 10% (*xmap*-promote-seq, PR) of the corresponding mmap percentage, and on average across all workloads from ~41% (*xmap*-huge-always, 18.73%) to ~22% (*xmap*-promote-seq, 9.91%). Finally, *xmap* with huge pages maintains the good scalability of *FastMap*. The average system time percentage across *xmap* huge-page configurations and Ligra workloads is 14.2%, compared to 11.3% for *xmap*-nohuge. At the same time, the average user time percentage is increased from 46% (*xmap*-promote-seq) to 119% (*xmap*-promote-always) compared to both mmap and *xmap*-nohuge.

Furthermore, the huge-page-based *xmap* configurations couple lower time spent in the kernel with efficient page writebacks which better utilize the available storage throughput. Figure 5.3 shows average I/O related statistics per configuration across all evaluated workloads. The Linux mmap achieves 437MB/s average write throughput and *xmap*-nohuge reaches 303MB/s. In contrast, with huge pages the average write throughput is increased from 23% (*xmap*-promote-seq, 536MB/s) up to 83% (*xmap*-huge-always, 799MB/s) compared to mmap, and from 77% up to 164% compared to *xmap*-nohuge. Simultaneously, even with fine-tuning the

Linux mmap incurs ~90% increased write traffic with 338GB on average compared to 172-175GB across *xmap* configurations. With respect to average read traffic, the huge page size is equal to the maximum readahead unit for Linux mmap. Therefore the *xmap* huge page configurations incur read traffic higher by 1.69× (*xmap*-promote-seq, 497GB) to 3.13× (922GB, *xmap*-huge-always) than the 295GB of mmap, depending on how eagerly huge pages are used. The Linux mmap itself reads 6% more from storage compared to *xmap*-nohuge (277GB) due to page readahead. The read traffic penalty of huge pages is offset by the higher read throughput achieved through large sequential reads for huge pages. Linux mmap is able to issue average reads of 1.3MB thanks to its readahead, which is 24× larger than *xmap*-promote-seq (51KB), 1.7× larger than *xmap*-promote-always (729KB) and 1.65× lower than *xmap*-huge-always(2MB). The relationship between *xmap* huge page configurations and average read size is dependent on huge-page-sized reads on all page faults (*xmap*-huge-always), more frequent promotions (*xmap*-promote-always), or less frequent promotions (*xmap*-promote-seq). Regardless of frequency, huge-page-sized sequential reads result in higher average read throughput than the vectored page I/O employed by mmap's readahead, from 2× (*xmap*-promote-seq, 1.1GB/s) to 5× (*xmap*-huge-always, 2.8GB/s) compared to mmap (556MB/s).

Impact of individual *xmap* mechanisms: First, we examine the improvement of adding static huge pages (*xmap*-huge-always) without asynchronous promotions. *xmap*-huge-always outperforms both Linux mmap and *xmap*-nohuge in all Ligra workloads in terms of overall execution time (Figure 5.1), and is on average 84% faster than the former and 123% faster than the latter. It additionally retains the good scalability of *FastMap* (18.08% average system time, Figure 5.2) and increases average user time percentage to 22.43%, compared to 12.51% and 12.53% for mmap and *xmap*-nohuge, respectively. However, the increased I/O cost associated with synchronously preparing a huge page at page fault time is reflected in the increased average read traffic, shown in Figure 5.3, which at 921.61GB is 3.33× higher than the "expected minimum" 276.52GB with *xmap*-nohuge -as it uses the minimum page size with no readahead and a common page eviction policy. This is also reflected in the increased average IO-wait time percentage for *xmap*-huge-always (38.1%), that is still only 10% higher than the corresponding percentage for *xmap*-nohuge (34.6%), which synchronously performs 512× smaller read I/Os on page faults. We can attribute the IO-wait penalty being this small to the fact that huge pages drastically reduce major page faults that require synchronous I/O in the first place, by 192× compared to *xmap*-nohuge (Table 5.1). Conversely, the average write traffic across all workloads (Figure 5.3) is similar for both *xmap*-huge-always (172GB) and *xmap*-nohuge (173GB), which is a good indicator that Ligra fully utilizes huge pages, as it shows that dirty huge pages being written back to storage are modified in their entire span.

Next, we examine the additional impact of introducing asynchronous huge page promotions with *xmap*-promote-always, which further reinforces the benefits

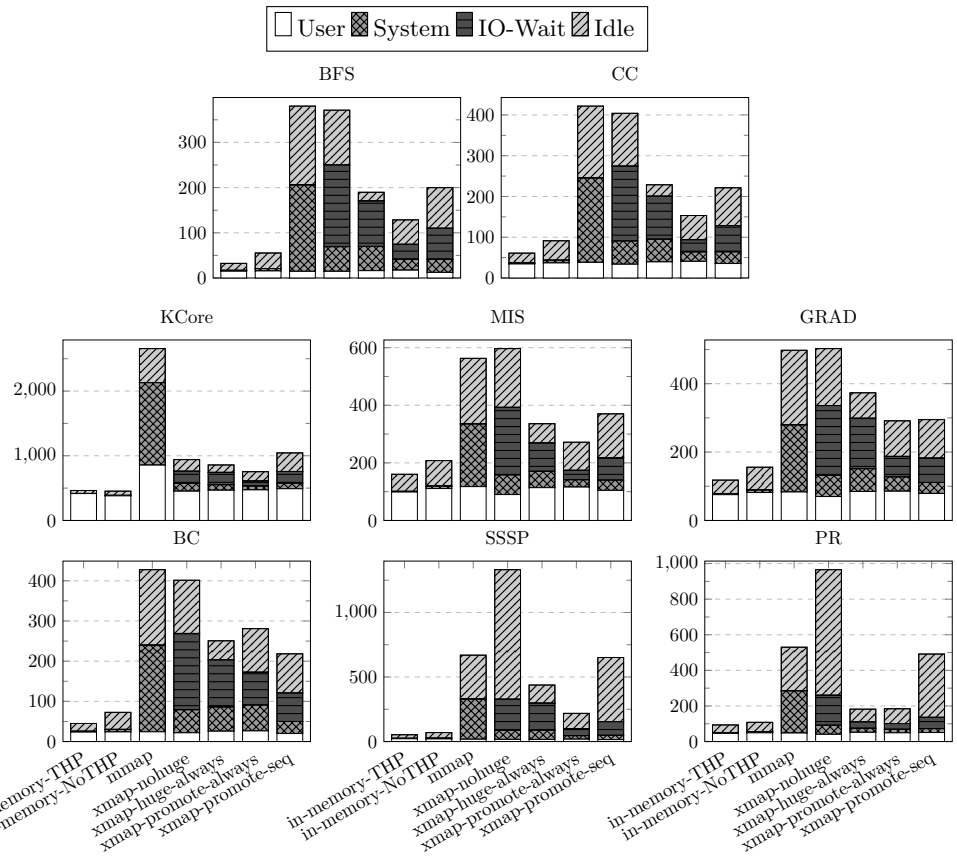

Figure 5.2: Ligra execution time breakdown. The y-axis represents absolute execution time in seconds.

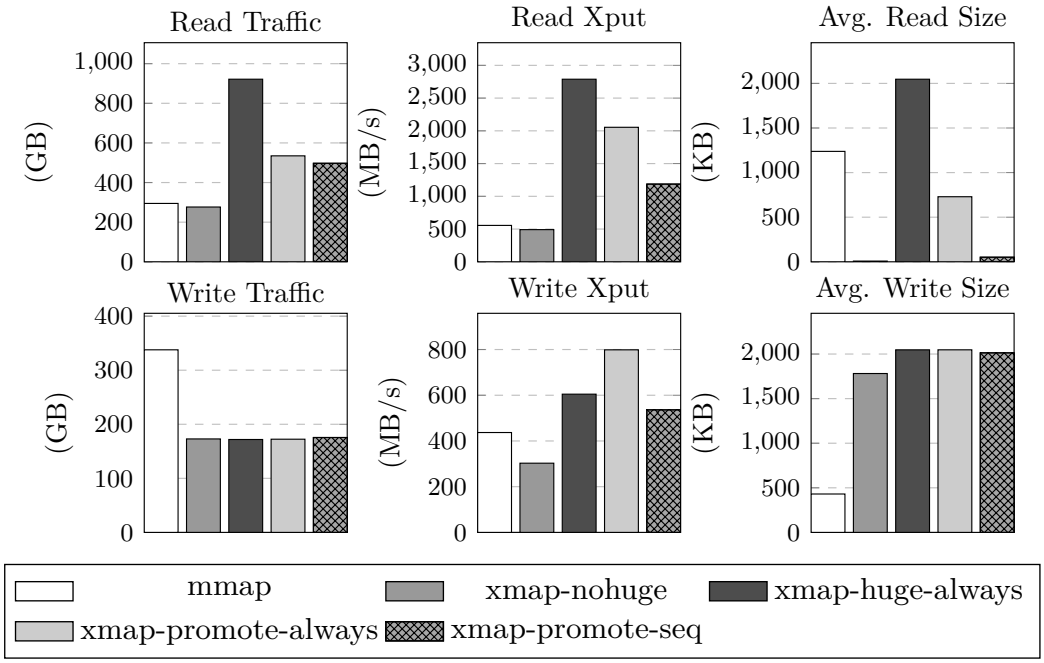

Figure 5.3: Average I/O statistics across all Ligra workloads.

introduced by huge pages. On the basis of total execution time, *xmap*-promote-always improves performance over *xmap*-huge-always on 6 of 8 workloads, and the difference on the two other workloads (KCore, BC) is negligible enough that *xmap*-promote-always incurs a 3.08× performance penalty on average, 2.37× better than the Linux mmap. Especially noteworthy is the SSSP workload, where *xmap*-promote-always incurs a 4.12× performance penalty, compared to 14.57× on average for the rest of the out-of-core configurations (11.06× if we exclude the *xmap*-nohuge outlier).

The execution time breakdowns in Figure 5.2 show that asynchronous promotions limit page fault latency, leading to 22% higher average user time (27.41% of execution time to 22.43%) and 48% lower IO-wait time (20% of execution time to 38.1%), compared to *xmap*-huge-always. Furthermore, asynchronous promotions alleviate page allocation pressure from the huge page pool as follows. In *xmap*-huge-always, concurrent page faults to page offsets within a VMA range, race to allocate a huge page for neighbouring VMA addresses. Eventually, only one will succeed in inserting a huge page to the page cache tree. This causes clean huge page evictions because the workload dataset does not fit in memory, which, coupled with the coarser granularity of huge pages, causes more major page faults in the future. Instead, with asynchronous promotions, concurrent page faults allocate regular pages at finer grain from the regular page pool and then race to reserve a promotion in the corresponding reservation list. Only one page fault succeeds in

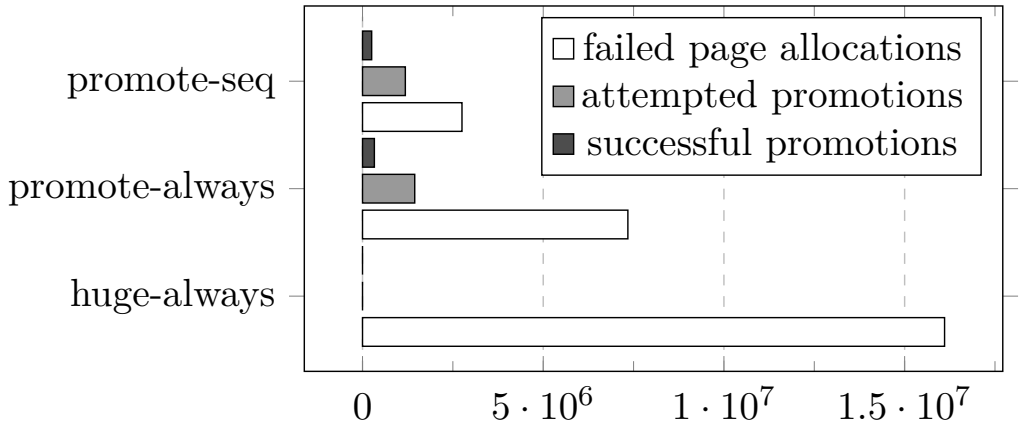

Figure 5.4: Average statistics related to huge pages across all Ligra workloads for three *xmap* configurations. Attempted promotions indicate page faults which attempted to scheduled a huge page promotion. Successful promotions indicate page faults which successfully created a promotion reservation, eventually leading to a completed promotion.

scheduling the promotion and only one huge page allocation is attempted from the huge page pool. The benefits are two-fold for *xmap*-promote-always compared to *xmap*-huge-always: It performs 1.72× less read traffic (Figure 5.3) and 2.2× fewer failed huge page allocations on average across all workloads (Figure 5.4).

Finally, we also examine a simple promotion policy based on page faults to consecutive virtual pages. *xmap*-promote-seq limits the total amount of attempted promotions by 18% compared to *xmap*-promote-always (Figure 5.4), thus further reducing average system time percentage on total execution to from 13.95% to 9.91%. This stems from the fact that asynchronous promotions are served by kernel threads. Triggering more asynchronous promotions requires kernel threads to be scheduled to perform them. If the application threads already saturate the available system cores then they are more likely to be involuntarily context switched by promotion kernel threads, leading to higher system time. Additionally, both *xmap*-promote-seq and *xmap*-promote-always exhibit similar percentage of successful over attempted promotions, (21% and 23%, respectively), further indicating that Ligra has a highly sequential access pattern because promotions are triggered at similar rates. However, with *xmap*-promote-seq the delay in scheduling a promotion triggers more total page faults (19.5× compared to *xmap*-promote-always, Table 5.1) which are served with regular pages and become detrimental to performance. Thus *xmap*-promote-seq only outperforms *xmap*-huge-always and *xmap*-promote-always on the GRAD and BC workloads, and incurs 4.91× penalty on execution time on average. Nevertheless, even a trivial promotion policy such as *xmap*-promote-seq outperforms the regular page based mmap by 1.5× and *xmap*-nohuge by 1.84× in terms of average performance penalty (Figure 5.1). Moreover,

xmap-promote-seq has the lowest average incurred read traffic (497GB, Figure 5.3) among all huge-page-using configurations, 1.85× lower than *xmap*-huge-always and 1.07× lower than *xmap*-promote-always. Thus, such a policy may be beneficial if I/O traffic is also important besides absolute performance, e.g., for device lifetime considerations.

Chapter 6

Related Work

We divide related work in three categories: a) *mmio*, b) huge-page support, and c) I/O memory and cache management.

***mmio*:** DI-MMAP [VE+15] is a Linux kernel module which improves the performance of *mmio* in Linux for out-of-core, data-intensive applications. It uses a page pool architecture and organizes pages in FIFO queues. Similarly, the authors in [Son+16] propose modifications to the Linux kernel to improve *mmio* performance over fast storage devices. The authors in [CKH17] aim to improve *mmio* performance for in-memory filesystems over NVM using: an asynchronous preemptive page-table entry creation mechanim, a cache for recently unmapped virtual memory areas, and extensions to the Linux *madvise* mechanism. UMap [Pen+22] is a user-space memory mapping library, implemented in Linux using the *userfaultfd* user-space fault handler. The authors note the lack of huge page support for file-backed mappings in Linux as a motivating factor for the shift to user-space, which incurs additional software overhead. UMap organizes data in regions for the purpose of scalability which bear similarities to the page pool architecture of *xmap*. *xmap* is the first system to provide extensive support file-backed huge pages, asynchronous operations, and to offer more control over the I/O path. *FastMap* [Pap+20a] is an alternative *mmio* path implemented as a Linux kernel module, aimed at addressing scalability limitations in the current Linux *mmap* implementation. *xmap* builds on *FastMap* by adding support for NUMA aware allocations, explicit *buffered I/O* and *direct I/O* operations, arbitrary number of reverse mappings, VMA splitting, transparent file-backed huge pages and asynchronous operations. Aquila [PMB21] is a library OS compatible with Linux *mmap*, which places applications in non-root ring 0 in order to combine the fast cache hit path of *mmio* with the capability for customization of the I/O path offered by user-space I/O cache approaches. Aquila, contrary to *xmap*, requires application modifications and does not provide support for huge pages. Park et al. [PKS13] propose an approach where dirty pages are written atomically to the underlying filesystem. This *atomic msync* approach is a coarse grain form of maintaining update order

and is orthogonal to *xmap*.

Huge-page mechanisms: The Linux kernel implements two mechanisms for the utilization of huge pages. *Transparent Huge Pages* (THP) [Docb], which allows applications to transparently use huge pages for *anonymous mappings,* which involve no I/O. Ingens [Kwo+16] and HawkEye [PBG19] build on the Linux THP mechanism. Both approaches only address anonymous mappings as well. They aim to mitigate memory fragmentation problems stemming from the aggressive huge page allocation policy of *khugepaged*. Ingens tracks baseline page utilization and access frequency with bit-vectors to promote or preemptively demote huge pages. HawkEye tracks via hardware counters or estimates virtual address translation overheads stemming from TLB misses and uses this information along with the overall utilization of an anonymous huge page sized area to schedule promotions and demotions. Other Linux-based work related to huge pages attempts to address memory fragmentation issues associated with huge page allocations at the OS memory management level [GH08; PPG18], or the user-space memory allocator level [Maa+21; Hun+21]. Finally, the authors in [Heo+22] explore the use of huge pages and page migration between fast DRAM and slower NVM based on adaptive policy selection for tiered memory systems. All of these works only concern themselves with anonymous huge pages. The second mechanism, *HugeTLB pages* [Doca], is a static huge page allocation mechanism. A kernel boot parameter can reserve an amount of memory for a kernel-space huge-page pool. Processes may then use these huge pages by mounting the pseudo-filesystem *hugetlbfs*. Any "file" created within this pseudo-filesystem will statically use huge pages from the pool. It is important to note that there is no promotion or demotion involved with these pages and they cannot be swapped out under memory pressure [GH10]. *xmap* provides extensive support for using huge-pages with file-backed *mmio*.

FreeBSD supports transparent huge pages for both file-backed and anonymous mappings for x86 and x86_64 [Pro09] (since version 7.2-RELEASE) and for ARM [Pro14] (since version 10.0-RELEASE). Their approach is based on a huge page reservation mechanism and supports promotions and demotions. This mechanism was first introduced and analysed by Navarro et al. [Nav+02]. Quicksilver [ZCR20] is a huge page management mechanism which builds on the FreeBSD support for huge pages. It relaxes FreeBSD's promotion requirements by preparing and promoting a huge page when the number of faults on regular subpages exceeds a threshold. It preemptively deallocates huge page reservations if they are underutilized to improve availability of huge pages for allocation. *xmap* offers applications transparent support for file-backed huge pages similar to the mechanisms available in FreeBSD, however it places larger focus on asynchronous mechanisms for huge page promotion and page prefetching, whereas FreeBSD incrementally prepares and promotes huge pages synchronously at page fault time. Windows [YSI17] supports huge pages for all types of mappings. An application requests the use huge pages via a flag, *MEM_LARGE_PAGES*. Windows statically

allocates huge pages on every page fault, with the risk of application failure if not enough contiguous memory is found for a huge page. Unlike *xmap*, no promotion or demotions are available.

I/O memory and cache management: TMO [Wei+22] is a set of Linux kernel mechanisms along with a user-space framework to monitor resource (i.e., CPU, memory and I/O) pressure and offload memory to either flash-based storage or a compressed in-memory swap, in order to save memory and improve workload performance at datacenter-scale. *xmap* is orthogonal to TMO, which could provide directives to *xmap* for modifying at runtime the size of the memory pools. DAOS [PBU22] is a memory management system consisting of a kernel-level data access monitor DAMON and a user-space framework which accepts and automatically tunes user-defined schemas. One of the actions implemented by DAOS is an *MADV_HUGEPAGE* system call (via madvise) to instruct the kernel to use huge pages. *xmap* can be combined with this feature of DAOS to provide a workload-driven, dynamic huge page selection/promotion policy. We leave this as future work. TriCache [Fen+22], is a user-space, transparent block-cache designed for scalable out-of-core computation over arrays of SSDs. It requires compile-time instrumentation of application code through LLVM to translate load/store instructions into TriCache calls. The block cache itself uses a layered address translation mechanism and SPDK to communicate with the underlying devices. *xmap* shares similar goals, however, uses the kernel *mmio* path, which is inherently more transparent, eliminates all I/O cache hit costs, and additionally allows for efficient memory sharing between processes, which as the authors themselves note is not possible with TriCache. SPDK [Int] is a set of Intel developed libraries for the direct control of NVMe SSD devices from user-space. SPDK aims to reduce the overhead of *direct I/O* system calls. *xmap* transparently reduce overhead in the I/O path for applications that use *mmio*.

Chapter 7

Future Work & Conclusions

Our current research and future work into *xmap* is focused around: (a) Further improving the concurrency of the asynchronous huge page promotion mechanism with regard to application execution. (b) Implementing dynamic self-tuning policies for huge page selection and promotion scheduling. (c) Transparently implementing *madvise*-based I/O path control semantics beyond MADV_(NO)HUGEPAGE, such as MADV_WILLNEED and MADV_DONTNEED. (d) Applying and evaluating said semantics to storage caching applications such as key-value stores.

File-backed *mmio* has the potential to allow applications to transparently extend their heap over fast storage devices, without the need for extensive application redesign and to eliminate overhead for cache hits in I/O caches. In this thesis we present *xmap*, a next-generation design for file-backed *mmio* in Linux. *xmap* addresses three important limitations of *mmap*, the current *mmio* implementation: (a) It improves scalability with the number of cores, (b) it adds extensive support for huge pages, and (c) it provides support for asynchronous operations. We implement and evaluate *xmap* in Linux. We analyze the potential of *madvise* based I/O path control via *xmap* and evaluate the impact of *xmap* for transparently extending the application heap over fast storage devices. To this end we use Ligra, a graph processing framework, with a variety of graph workloads and with no code modifications. Our results show that, compared to Linux mmap, *xmap* enables up to 3.5× faster out-of-memory execution, reduces total page faults by up to 265×, decreases CPU system time percentage by up to 90%, and increases CPU user time percentage by up to 250% processing graph datasets 6-8× larger than the available system DRAM.

Bibliography

[ATW20] Nadav Amit, Amy Tai, and Michael Wei. "Don't Shoot down TLB Shootdowns!" In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387518. URL: https://doi.org/10.1145/3342195.3387518.

[BM04] Sorav Bansal and Dharmendra S. Modha. "CAR: Clock with Adaptive Replacement." In: *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*. FAST '04. San Francisco, CA: USENIX Association, 2004, pp. 187–200.

[Cao+20] Zhichao Cao et al. "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook." In: *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 209–223. ISBN: 978-1-939133-12-0. URL: https://www.usenix.org/conference/fast20/presentation/cao-zhichao.

[Cha+08] Fay Chang et al. "Bigtable: A Distributed Storage System for Structured Data." In: *ACM Trans. Comput. Syst.* 26.2 (2008). ISSN: 0734-2071. DOI: 10.1145/1365815.1365816. URL: https://doi.org/10.1145/1365815.1365816.

[CKH17] Jungsik Choi, Jiwon Kim, and Hwansoo Han. "Efficient Memory Mapped File I/O for In-Memory File Systems." In: *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. Santa Clara, CA: USENIX Association, July 2017. URL: https://www.usenix.org/conference/hotstorage17/program/presentation/choi.

[CKZ12] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. "Scalable address spaces using RCU balanced trees." In: *ACM SIGPLAN Notices* 47.4 (2012), pp. 199–210.

[CKZ13] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. "RadixVM: Scalable Address Spaces for Multithreaded Applications." In: *Proceedings of the 8th ACM European Conference on Computer*

Systems. EuroSys '13. Prague, Czech Republic: Association for Computing Machinery, 2013, pp. 211–224. ISBN: 9781450319942. DOI: 10.1145/2465351.2465373. URL: https://doi.org/10.1145/2465351.2465373.

[Cor] Jonathan Corbet. "An introduction to compound pages." In: ().

[Cor03] Jonathan Corbet. "The object-based reverse-mapping VM." In: (2003).

[Cor04] Jonathan Corbet. "Virtual Memory II: the return of objrmap." In: (2004).

[DeC+07] Giuseppe DeCandia et al. "Dynamo: Amazon's highly available key-value store." In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[Doca] Linux Kernel Documentation. *HugeTLB Pages*. https://www.kernel.org/doc/html/latest/admin-guide/mm/hugetlbpage.html.

[Docb] Linux Kernel Documentation. *Transparent Hugepage Support*. https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html.

[Fen+22] Guanyu Feng et al. "TriCache: A User-Transparent Block Cache Enabling High-Performance Out-of-Core Processing with In-Memory Programs." In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 395–411. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/feng.

[GH08] Mel Gorman and Patrick Healy. "Supporting Superpage Allocation without Additional Hardware Support." In: *Proceedings of the 7th International Symposium on Memory Management*. ISMM '08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 41–50. ISBN: 9781605581347. DOI: 10.1145/1375634.1375641. URL: https://doi.org/10.1145/1375634.1375641.

[GH10] Mel Gorman and Patrick Healy. "Performance Characteristics of Explicit Superpage Support." In: *Proceedings of the 2010 International Conference on Computer Architecture*. ISCA'10. Saint-Malo, France: Springer-Verlag, 2010, pp. 293–310. ISBN: 9783642243219. DOI: 10.1007/978-3-642-24322-6_24. URL: https://doi.org/10.1007/978-3-642-24322-6_24.

[Gro] Persistent Memory Programming Group. *Volatile Persistent Memory Allocator*. https://github.com/pmem/vmem/.

[Heo+22] Taekyung Heo et al. "Adaptive Page Migration Policy With Huge Pages in Tiered Memory Systems." In: *IEEE Transactions on Computers* 71.1 (2022), pp. 53–68. DOI: 10.1109/TC.2020.3036686.

[Hun+21] A.H. Hunter et al. "Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator." In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021, pp. 257–273. ISBN: 978-1-939133-22-9. URL: https://www.usenix.org/conference/osdi21/presentation/hunter.

[Int] Intel Corporation. *Storage Performance Development Kit*. https://spdk.io/.

[Kol+20] Iacovos G. Kolokasis et al. "Say Goodbye to Off-Heap Caches! On-Heap Caches Using Memory-Mapped I/O." In: *Proceedings of the 12th USENIX Conference on Hot Topics in Storage and File Systems*. HotStorage'20. USA: USENIX Association, 2020.

[Kwo+16] Youngjin Kwon et al. "Coordinated and Efficient Huge Page Management with Ingens." In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 705–721. ISBN: 9781931971331.

[LC22] Karol Latecki and Jaroslaw Chachulski. *SPDK NVMe BDEV Performance Report Release 22.05*. Tech. rep. Intel Corporation, 2022. URL: https://ci.spdk.io/download/performance-reports/SPDK_nvme_bdev_perf_report_2205.pdf.

[Lin] Linux Kernel Documentation. *Memory Resource Controller*. URL: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/memory.html.

[Maa+21] Martin Maas et al. "Adaptive Huge-Page Subrelease for Non-Moving Memory Allocators in Warehouse-Scale Computers." In: *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*. ISMM 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 28–38. ISBN: 9781450384483. DOI: 10.1145/3459898.3463905. URL: https://doi.org/10.1145/3459898.3463905.

[MM03] Nimrod Megiddo and Dharmendra S. Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache." In: *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*. FAST '03. San Francisco, CA: USENIX Association, 2003, pp. 115–130.

[Nav+02] Juan Navarro et al. "Practical, Transparent Operating System Support for Superpages." In: *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (Copyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading)*. OSDI '02. Boston, Massachusetts: USENIX Association, 2002, pp. 89–104. ISBN: 9781450301114.

[net] Friendster social network. *Friendster: The online gaming social network*. URL: https://archive.org/details/friendster-dataset-201107.

[Pap+16] Anastasios Papagiannis et al. "Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store." In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, June 2016, pp. 537–550. ISBN: 978-1-931971-30-0. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/papagiannis.

[Pap+20a] Anastasios Papagiannis et al. "Optimizing Memory-Mapped I/O for Fast Storage Devices." In: *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-14-4.

[Pap+20b] Anastasios Papagiannis et al. "Optimizing Memory-mapped I/O for Fast Storage Devices." In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 813–827. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/papagiannis.

[PBG19] Ashish Panwar, Sorav Bansal, and K. Gopinath. "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 '19. Providence, RI, USA: Association for Computing Machinery, 2019. ISBN: 9781450362405.

[PBU22] SeongJae Park, Madhuparna Bhowmik, and Alexandru Uta. "DAOS: Data Access-Aware Operating System." In: *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '22. Minneapolis, MN, USA: Association for Computing Machinery, 2022, pp. 4–15. ISBN: 9781450391993. DOI: 10.1145/3502181.3531466. URL: https://doi.org/10.1145/3502181.3531466.

[Pen+22] Ivy B. Peng et al. "Enabling Scalable and Extensible Memory-Mapped Datastores in Userspace." In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 866–877. DOI: 10.1109/TPDS.2021.3086302.

[PKS13] Stan Park, Terence Kelly, and Kai Shen. "Failure-Atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data." In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: Association for Computing Machinery, 2013, pp. 225–238. ISBN: 9781450319942. DOI: 10.1145/2465351.2465374. URL: https://doi.org/10.1145/2465351.2465374.

[PMB21] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. "Memory-Mapped I/O on Steroids." In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys '21. Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 277–293. ISBN: 9781450383349. DOI: 10.1145/3447786.3456242. URL: https://doi.org/10.1145/3447786.3456242.

[PPG18] Ashish Panwar, Aravinda Prasad, and K. Gopinath. "Making Huge Pages Actually Useful." In: *SIGPLAN Not.* 53.2 (2018), pp. 679–692. ISSN: 0362-1340. DOI: 10.1145/3296957.3173203. URL: https://doi.org/10.1145/3296957.3173203.

[Pro09] The FreeBSD Project. *FreeBSD 7.2-RELEASE Release Notes*. https://www.freebsd.org/releases/7.2R/relnotes-detailed/. 2009.

[Pro14] The FreeBSD Project. *FreeBSD 10.0-RELEASE Release Notes*. https://www.freebsd.org/releases/10.0R/relnotes/. 2014.

[RA15] Ryan A. Rossi and Nesreen K. Ahmed. "The Network Data Repository with Interactive Graph Analytics and Visualization." In: *AAAI*. 2015. URL: https://networkrepository.com.

[SB13] Julian Shun and Guy E. Blelloch. "Ligra: A Lightweight Graph Processing Framework for Shared Memory." In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '13. Shenzhen, China: Association for Computing Machinery, 2013, pp. 135–146. ISBN: 9781450319225. DOI: 10.1145/2442516.2442530. URL: https://doi.org/10.1145/2442516.2442530.

[Son+16] Nae Young Song et al. "Efficient Memory-Mapped I/O on Fast Storage Device." In: *ACM Trans. Storage* 12.4 (2016). ISSN: 1553-3077. DOI: 10.1145/2846100. URL: https://doi.org/10.1145/2846100.

[VE+15] Brian Van Essen et al. "DI-MMAP—a scalable memory-map runtime for out-of-core data-intensive applications." In: *Cluster Computing* 18.1 (2015), pp. 15–28.

[Wei+22] Johannes Weiner et al. "TMO: Transparent Memory Offloading in Datacenters." In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '22. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 609–621. ISBN: 9781450392051. DOI: 10.1145/3503222.3507731. URL: https://doi.org/10.1145/3503222.3507731.

[YSI17] Pavel Yosifovich, David A Solomon, and Alex Ionescu. *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more*. Microsoft Press, 2017.

[Zah+10] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets." In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, p. 10.

[ZB99] Erez Zadok and Ion Badulescu. "A stackable file system interface for Linux." In: *LinuxExpo Conference Proceedings*. Vol. 94. 1999, pp. 141–151.

[ZCR20] Weixi Zhu, Alan L. Cox, and Scott Rixner. "A Comprehensive Analysis of Superpage Management Mechanisms and Policies." In: *Proc. of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-14-4.