LLNL-JRNL-819817

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Enabling Scalable and Extensible Memory-mapped Datastores in Userspace

I. B. Peng, M. B. Gokhale, K. Youssef, K. Iwabuchi, R. Pearce

February 23, 2021

IEEE Transactions on Parallel and Distributed Systems

Enabling Scalable and Extensible Memory-mapped Datastores in Userspace

Ivy B. Peng, *Member, IEEE,* Maya B. Gokhale, *Fellow, IEEE,*
Karim Youssef, Keita Iwabuchi, Roger Pearce

Abstract—Exascale workloads are expected to incorporate data-intensive processing in close coordination with traditional physics simulations. These emerging scientific, data-analytics and machine learning applications need to access a wide variety of datastores in flat files and structured databases. Programmer productivity is greatly enhanced by mapping datastores into the application process's virtual memory space to provide a unified "in-memory" interface. Currently, memory mapping is provided by system software primarily designed for generality and reliability. However, scalability at high concurrency is a formidable challenge on exascale systems. Also, there is a need for extensibility to support new datastores potentially requiring HPC data transfer services. In this work, we present *UMap*, a scalable and extensible userspace service for memory-mapping datastores. Through decoupled queue management, concurrency aware adaptation, and dynamic load balancing, *UMap* enables application performance to scale even at high concurrency. We evaluate *UMap* in data-intensive applications, including sorting, graph traversal, database operations, and metagenomic analytics. Our results show that *UMap* as a userspace service outperforms an optimized kernel-based service across a wide range of intra-node concurrency by 1.22-$1.9\times$. We performed two case studies to demonstrate *UMap*'s extensibility. First, a new datastore residing in remote memory is incorporated into *UMap* as an application-specific plugin. Second, we present a persistent memory allocator *Metall* built atop *UMap* for unified storage/memory.

Index Terms—data-intensive, userspace, memory mapping, mmap, memory-mapped I/O.

1 Introduction

The emergence of analytics as a dominant component of supercomputing workloads has a profound influence on exascale system architecture. Traditional simulation workloads will rely on in-situ analytics to mitigate bandwidth limitations to accessing global storage. New exascale applications searching key/value stores, graphs, and scientific data stores, will need large capacity memories to hold analytics datasets [1], [2]. Technology drivers that enable data-driven workloads include fast persistent memory that introduces a new tier in the memory/storage hierarchy [3], [4] and extreme-scale parallelism in CPU and GPU architectures that offers new levels of on-node concurrency.

The rapidly evolving landscape of analytics-driven workloads, combined with the blurred boundary between memory and storage, and high-degree parallelism, offers an opportunity to shift from traditional file read/write I/O to a unified memory model in which data is accessed as if entirely in memory. When files stored in fast node-local storage are memory-mapped, an application is essentially provided with a unified storage/memory of significantly increased capacity. Data-intensive applications can exploit memory-mapped I/O to extend the virtual address space of a process. On the Summit and Lassen pre-exascale machines, node-local NVMe SSD can extend the capacity by terabytes. Accessing datasets in storage through memory mapping shifts the burden of paging, prefetching, and caching data between storage and memory to the operating system. Memory mapping is used in many system software components and user-level libraries. The widely used FITS (Flexible Image Transport System) library, used by the astronomy community to store images and tables, offers memory mapping. Many key/value stores, such as sqlite, lucene, couchbase/moss, provide mmap access. On future Exascale machines, even higher use of memory mapping is predicted as an efficient way to generate, share, and analyze application-specific binary format data. In the exascale regime, enabling scalable memory mapping at high concurrency becomes critical to sustaining the performance of these data-intensive applications. However, high concurrency in thread and process levels challenges the scalability of the whole software stack, including the memory mapping system software.

From the application perspective, knowledge gained by developers that has accumulated over many years of successful porting onto large-scale HPC machines can mitigate inefficiencies inherent in general purpose system software. Such application-specific knowledge can effectively steer optimization and improve performance portability using domain and algorithm information that is difficult to discern at the system or device level. For example, an in-transit I/O composition framework [5] can be customized by application developers to handle massive datasets embodying such structured objects as databases and key-value stores during

- *Ivy Peng is with Lawrence Livermore National Laboratory, CA, 94550. Email: peng8@llnl.gov*
- *Maya Gokhale is with Lawrence Livermore National Laboratory, CA, 94550. Email: gokhale2@llnl.gov*
- *Karim Youssef is with with Virginia Tech, Blacksburg, VA 24061. Email: karimy@vt.edu*
- *Keita Iwabuchi is with Lawrence Livermore National Laboratory, CA, 94550. Email: iwabuchi1@llnl.gov*
- *Roger Pearce is with Lawrence Livermore National Laboratory, CA, 94550. Email: pearce7@llnl.gov*

Manuscript received Feb 24, 2020.

the execution of a job. Instantiating datastores used in HPC applications as files in a storage system may introduce substantial inefficiency. For instance, data objects in a simulation process can be the data source for another analytics process in the same job. Currently, these new datastores cannot be memory-mapped but rely on application-specific customization of a data service. An extensible memory-mapping service would support the addition of new types of datastores while providing a unified interface to the application.

Existing system services like the Linux *mmap* can memory-map files or devices into the virtual memory of a process. This system service is most used in loading dynamic libraries and can also be used for memory extension to support out-of-core execution. However, system software is primarily designed for generality because it has to be tuned for performance reliability and consistency over a broad range of workloads. Thus, it might miss the opportunity of application-specific optimizations. Also, existing system service scales poorly at high concurrency [6]. Scalability at high concurrency is a unique challenge on exascale HPC systems when applications have to leverage high concurrency provided by the hardware, i.e., increased core count and hardware threads. Furthermore, as a system service, changes in configurations have system-wide impacts so that characteristically different applications have to compromise when co-running on one machine. Another unique challenge on Exascale machines comes from the dependence on the kernel. Users on HPC systems have little privilege in modifying kernel settings, which could be critical for performance optimization. How to support performance portability when running on different machines with different kernels? Finally, a kernel-based service also limits the extensibility of supporting new types of datastores to be memory-mapped into application processes' virtual address space.

The challenges in the system service motivate us to investigate a userspace service called *UMap* [7], a C++ library used by a (multi-threaded) application process. *UMap* employs the recent userfaultfd [8] mechanism to offload page fault handling from the kernel into userspace. In a userspace solution, special privileges are not needed to modify system parameters, as would be required by kernel-based solutions. It can also support much finer-granularity control than the kernel in concurrency, I/O granularity, caching, prefetching, and eviction policies to meet unique requirements in applications. When porting an application across different HPC systems, developers can communicate application knowledge through these controls to mitigate the impact from different kernels and reuse optimization efforts. *UMap* provides both API and environmental controls to enable configurable page sizes, eviction strategy, application-specific prefetching, and detailed diagnosis information to the programmer. The parameters configured in userspace are confined within the application so that conflicting choices in different applications can be supported in the same OS environment, which is infeasible through a system service. Finally, new types of data stores can be created for memory-map service through an abstraction layer without dependence on kernel support.

In this work, we prioritize scalability as the main design objective to optimize *UMap* for high concurrency in Exascale-ready applications. *UMap* employs decoupled queue management so that the fault handling process in the userspace is split into multiple stages. Each stage handles a simple task, i.e., polling notifications and fetching data from a datastore, and is assigned to a group of dedicated workers. The size of each worker group can be adapted based on the complexity of its assigned stage. Concurrency-aware adaptation is introduced to mitigate contention at high concurrency. At massive concurrency, the method for scheduling tasks from a task pool to workers is changed from single task to a bulk schedule, i.e., a batch of tasks to reduce synchronization on the shared data structure. Eviction also employs a non-blocking selection of victim pages to avoid waiting for specific page status. To mitigate contention from file systems for file-backed datastores, we introduce *SparseStore* that transparently splits a single backing file into multiple files for improved throughput. We observe that memory usage in highly concurrent applications could fluctuate considerably when the concurrency level changes. Hence, we introduce dynamic adaptation of the in-memory buffer capacity at runtime. Finally, when a large number of threads contend for a shared buffer, thrashing becomes severe. To address this, we introduce a hybrid caching policy in the buffer to support a combination of static and standard cache replacement.

In summary, our main contributions are as follows:

- We present the design and optimization of a userspace memory-mapping service scalable to the high **concurrency** level needed by Exascale applications.
- We provide an open-source implementation based on the *UMap* library[1].
- We evaluate the performance of *UMap* in data-intensive applications, including graph processing, database operations, data analytics, and a production metagenome analytic software.
- We demonstrate that *UMap* can outperform recent versions of the system service by up to 1.9 times at high concurrency.
- Through a sensitivity study, we show the impact of fine-grained, application-driven configurations through *UMap*.
- In two case studies, we illustrate *UMap* 's **extensibility**, namely a new datastore mapped to remote memory and a persistent memory allocator *Metall* used to construct and analyze very large dynamic graphs.

2 Background

Large data stores are memory-mapped in many data-intensive applications for memory extension beyond the capacity of physical memory. The most common approach is through file-backed *mmap* in Linux. An alternative is to use memory protection and signal handling. Due to the high overhead in signal handling, a more efficient approach is to leverage the asynchronous message mechanism through *userfaultfd*.

We compare the different paths for page fault handling in a file-backed mmap (step 2-5) and a userspace service

1. UMAP v2.0.0 https://github.com/LLNL/umap

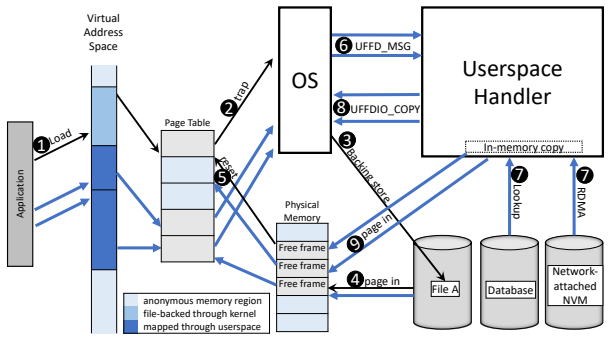


Fig. 1: We compare memory-mapping through the operating system with that through a userspace service.

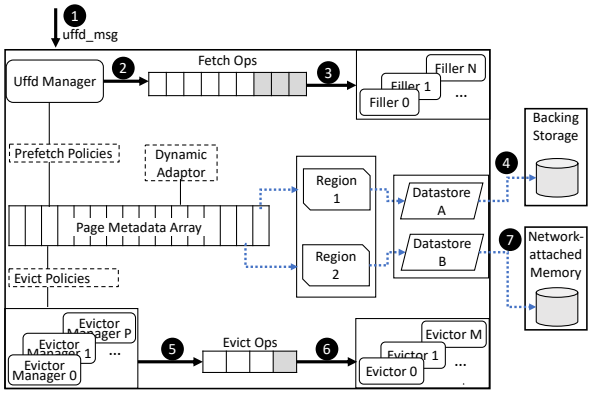


Fig. 2: An overview of the *UMap* architecture.

(step 2,6-10) in Figure 1. In the virtual address space, there is a file-backed *mmap* region (light blue) and two userfault regions (dark blue). The application accesses these memory regions through load and store instructions. A page fault is triggered if the page table entry (PTE) of the accessed address is not found in the page table. The page fault is trapped by the OS (step 2). If the OS finds the faulting address inside the file-backed memory region, it accesses the backing store, i.e., File A in this example, and copies the page into a free frame in the physical memory (step 4), and then resets the PTE in the page table (step 5). In contrast, in step 2, if the OS finds that the faulting address comes from the userfault memory regions, it will send a notification to the userspace handler (step 6). The userspace may support different types of data stores backing the userfault regions. In the example in Figure 1, the userspace service can fetch data either from a database or from remote network-attached memory (step 7). Once the userspace copies the required data into an in-memory buffer, it informs the OS through UFFDIO_COPY to atomically copy data in that buffer to a free frame in the physical memory, i.e., step 4 and 5 are common in both mmap and userfault paths.

The tradeoff between the kernel-based (i.e., mmap) and a userspace service (i.e., userfaultfd) lies in the overhead and flexibility. As shown in Figure 1, extra steps and communication are required in the userspace path, including step 6, 7, and 8. These result in increased latency of handling a page fault. Meanwhile, the kernel-based mmap often can bypass certain system calls and lower overhead [6]. However, only anonymous and file-backed memory-mapping are supported, so that path 7 to a remote memory page cannot be accomplished solely by the system service. Also, changing configurations of a system service often requires 'sudo' privilege, which is not permitted for most HPC users. Even the supported configurations are quite limited. For instance, anonymous memory regions support 4KB pages by default and may also support 2MB and 1GB pages with Linux's transparent huge page (THP). However, huge pages in file-backed memory regions are not supported in mainline kernels. A userspace service can provide many finer-granularity configurations than the system service. For instance, depending on the access pattern and compute intensity, an application may prefer a page size not supported by the system service. Moreover, a userspace service can be extended to support new types of data stores, such as databases and remote memory, without dependence on the kernel.

3 Design

The architecture of *UMap* is illustrated in Figure 2. Each userfault region in Figure 1 (dark blue) is represented by a *Region* object that records its start virtual address, size, and links to a backing data store object. A region is further divided into an array of uniform *UPages*, whose size can be configured at a fine granularity. *UMap* keeps a busy list of UPages whose data has been fetched into the DRAM and a free list of UPages that are yet to be assigned. By adjusting the total size of the busy and free lists, *UMap* can control the DRAM capacity it uses for caching. Data is physically located in a data store, which is defined in a *Datastore* object. *Datastore* provides an extensible abstraction layer so that new types of data stores can be defined. A *Datastore* must define functions for reading and writing data to their backing storage and may include additional functions for optimization and consistency. For instance, if files are used to represent a data store, the fetch functions could be implemented as read()/write() through a file system. If a remote-memory data store is defined, the functions could use RDMA or MPI for communication over the network. In the example of Figure 2, two types of data stores are defined, i.e., Datastore A for backing storage and Datastore B for network-accessed memory, respectively.

When page faults in the userfault regions occur, the OS sends notifications through *uffd_msg* to the userspace. The uffd manager is dedicated to polling for notifications and translating page faults into specific Fetch Ops (denoted as Fetch Ops in step 2). These Fetch Ops are distributed among a group of Filler workers (step 3) with configurable concurrency and load balancing (see Section 3.1). The filler workers are dedicated to fetching data from respective data stores into the physical memory. Eviction could occur either on-demand or proactively in the background. The Evict Managers are dedicated to selecting UPages that need to be evicted from the DRAM. They translate the selected victim UPages into specific Evict Ops (step 5), which are scheduled onto a group of *UMap Evictors* (step 6). We describe the design principles of *UMap* for scalability and extensibility in the remainder of the section.

3.1 Scalability

Scalability is achieved through concurrency optimizations and configurability.

3.1.1 Concurrency Adaptation

Decoupled Queue Management *UMap* splits the task of page fault handling into multiple stages, i.e., steps 1-7 in Figure 2), and supports concurrency adaptation in these stages. The number of workers used by a stage can be optimized according to the stage complexity, application characteristics, and hardware features. In general, short-latency stages only require a few workers while long-latency stages employ a large number of workers. For instance, polling notifications of page faults from the operating system (step 2 in Figure 2) is a simple task performed by one uffd manager. It translates a faulting event into specific Fetch Ops to be performed by workers in the next stage. In contrast, step 4 is a long-latency stage that retrieves data from external storage to the physical memory and thus, is often configured with the most workers to hide the long latency. Second, the concurrency in applications often affects the rates of generated page faults. When scaling up the concurrency in applications, the capability of handling page faults should also be scaled up accordingly. Such adaption to the application concurrency is supported in *UMap* but limited in the system service. Finally, the decoupled queue could also cope with the performance disparities in some storage devices. For instance, asymmetric read and write performance is a known issue in many persistent memories and SSD. Decoupling read and write operations into two separate groups, i.e., Fillers and Evictors could match the characteristics of different data stores.

Constrained Contention. We limit concurrent updates on shared data structures within a group and between groups to mitigate the synchronization overhead at high concurrency. Fillers and Evictors are the two largest groups of workers. Each Filler and Evictor is assigned with distinct UPages to work on, e.g., fetching or writing data and updating status. Therefore, even with a large number of Fillers and Evictors, the contention on shared resources is limited and does not increase appreciably with the number of workers. The busy list of UPages may be modified by the uffd manager and the Evict manager concurrently. We limit the contention on the busy list in two ways. First, the uffd manager appends entries to the tail while the Evict manager removes entries from the head. They do not clash on entries as long as the list is not empty. Second, only one uffd manager and few Evict managers are used because the tasks they need to perform are simple and fast. Finally, the free list can be updated by the uffd manager and the Evictors, i.e., the uffd manager removes while Evictors add entries to the free list. Hence, synchronization on the free list is needed to ensure correctness. As these two groups update the head and tail of the free list respectively, the contention is constrained as long as the free list is not empty. *UMap* does not build per-core lists as [6] because a userspace service does not assume that the application will always use all cores on a system.

Load balancing *UMap* manages all UPages that have been fetched in the physical memory in all userfaultfd memory regions in one pool so that the DRAM resource is allocated fairly based on the hotness of each region. Skewed data access is a common characteristic in data-intensive applications. Thus, different regions may differ significantly in the number of accesses. For instance, in graph processing, a graph often has regions of high degree vertices that result in more accesses than other regions. To address imbalanced accesses, Ops from all memory regions are pooled in one queue and redistributed among all workers. Load balancing is achieved through the dynamic scheduling in step 4 and 6, where each worker is assigned a uniform workload. Memory regions with more page faults will naturally be assigned with more workers because Ops in their region occupy larger portions in the pool.

Concurrency Control. High concurrency often imposes pressure on task scheduling, hardware, and memory usage. To tackle these challenges, *UMap* employs concurrency controls. First, when scheduling Ops in a queue to a group of workers, i.e., step 4 and 6, *UMap* changes to a batch schedule if the scheduling overhead outweighs the cost of completing an individual work item. The metric for switching the scheduling policy is defined by the ratio between the number of dispatched work items and inactive workers in a sampling period. A low metric value indicates too few items get scheduled while a large number of workers are waiting for assignment. Once switched, a batch of work items will be assigned to a worker at one time. The batch size is empirically set to 16. To accommodate a batch schedule, we also change the victim page selection to a batch selection. Modifying the normal LRU policy that selects the least recently used page for eviction, *UMap* selects a batch of least recently used pages that are available without the need for waiting for writing back or other operations to complete. Note that reducing the number of workers cannot address this problem because the scheduling overhead is not amortized and also the concurrency in this stage would be reduced. Second, some devices require concurrency control to avoid performance degradation. For instance, the Optane DC persistent memory can sustain high read bandwidth at high concurrency but write bandwidth degrades substantially at high concurrency [9]. Through concurrency control, *UMap* can throttle the number of threads performing concurrent writes to avoid performance degradation.

Finally, memory usage fluctuates in a running system when an application changes phases and when other processes start or finish. We support the dynamic adaptation of DRAM usage in *UMap* to address these challenges. A background thread in *UMap* monitors memory usage in *UMap* and available memory on the system. Adaptation is performed to either reduce memory usage in *UMap* when available memory on the system is low or to increase *UMap* memory when more system memory becomes available. The monitoring overhead is negligible because it is off the critical path. However, the adaptation has overhead from synchronization and reallocation. We reduce the overhead of adaptation by reducing its occurrence to the minimum possible based on estimation. In particular, *UMap* keeps a history of the number of processed page faults and the change in memory usage in *UMap* in the last N periods. If the memory usage in *UMap* is estimated to exceed the available memory on the system in the next M periods,

UMap reduces the size of the free list accordingly. Instead, if the free list is predicted to be exhausted in the next M periods while the available memory on the system is substantially larger, *UMap* increases the size of the free list. A larger N keeps a longer history but is less reactive to new changes. A large M results in unnecessary adaptations to occur early. Empirically, we set N to be five and M to be ten.

3.1.2 Configurability

One unique advantage of HPC applications is the accumulated application knowledge gained through many years of porting efforts towards Exascale machines. *UMap* exposes extensive fine-grained controls to the userspace to leverage such application knowledge, which is either not possible or difficult to accomplish in kernel-based approaches.

DRAM Usage. *UMap* supports specific constraints of DRAM usage for caching memory-mapped regions. For multiple applications co-running on a compute node, their DRAM usage for *UMap*-ed regions can be allocated based on their relative priorities. Currently, the system service cannot limit the DRAM usage directly, and an application's performance may fluctuate due to interference from other applications' activities. Also, applications exhibit different sensitivity to page cache size, i.e., increasing page cache may not bring comparable performance improvement, which also requires explicit configurations of DRAM usage.

I/O Granularity. The size of a UPage is configurable and it determines the finest I/O granularity in memory-mapping. Linux systems typically support limited page sizes, i.e., 4K, 2M, 1G, in anonymous memory regions, and only 4K page size is supported in file-backed memory regions. The optimal I/O granularity in an application often depends on the access patterns and compute intensity. With fine-grained control, an application's page size can be configured to better match its access profile.

Concurrency. The number of workers in the stages of page fault handling can be configured separately. The changes in concurrency can be either tuned for architecture-aware optimization or based on application-specific knowledge. For instance, the number of fillers and evictors can be adjusted to the read and write performance of the backing storage, respectively. Through *UMap*, architecture- and application-specific optimizations require no modifications to the kernel and only impact the application. Multiple applications can co-exist in the same OS with their unique configurations.

Hybrid Caching. *UMap* supports a hybrid of static and dynamic cache policies for memory-mapped regions. When evicting pages from the physical memory, LRU cache replacement policy is employed by default. However, at high concurrency, when a large number of threads access the same hot memory region, thrashing can become severe, causing frequent page-in and out of the same pages. To address this, *UMap* supports static pinning of partial memory regions in the physical memory, while employing LRU cache replacement for the other parts. The selection of static partitions needs to leverage application knowledge, e.g., application algorithms or offline profiling, to identify subregions that are densely accessed throughout the execution. While pinning pages is possible in the system service, there are system level limits on how much memory an unprivileged process may lock. *UMap* imposes no such limits. *UMap* also supports irregular prefetching that brings selected parts of memory regions into the physical memory asynchronously in the background.

4 Extensibility

In this section, we illustrate the extensibility of *UMap* in supporting two new types of data stores – SparseStore and RemoteStore.

4.1 SparseStore

SparseStore is a backing store optimization that improves storage efficiency of working with sparse data. *SparseStore* also enhances the scalability of multi-threaded applications by reducing I/O contention on the backing store.

SparseStore partitions the backing file of a *UMap* region into multiple files of equal size. Correspondingly, the *UMap* region is logically partitioned into segments, where each segment maps to a backing file. The backing files are created on-demand, i.e., when an *UMap* region is allocated, no backing files are created until their mapped segments in the *UMap* region are written for the first time. The on-demand creation of backing files provides storage efficiency for sparse data, while the multi-file solution reduces the multi-threaded contention on the backing store as threads writing to different files are not serialized. *SparseStore* is implemented as a store object in *UMap* that can optionally be used instead of the default store object. A *SparseStore* object is instantiated in either `create` mode when mapping an empty region, or `open` mode when mapping a region that was previously created using *SparseStore*. The segment granularity is a configurable parameter that can be set when instantiating a *SparseStore* object in `create` mode. *SparseStore* implements `read()` and `write()` backing store access functions that are invoked by *UMap* when handling a page fault. These functions map the page address to a backing file index and file offset. If a dirty page is being evicted for the first time, the `write()` function will create the corresponding file and then writes the page to its corresponding file offset.

4.2 RemoteStore

RemoteStore is a new data store that uses the physical memory of remote compute nodes as the backing storage. The main motivation for *RemoteStore* is to understand the impact of future HPC systems that are equipped with heterogeneous sub-clusters, e.g., memory nodes with huge memory capacity and slim nodes with small memory capacity. Clusters equipped with memory nodes and slim nodes can achieve high memory utilization for compute-intensive workloads and still meet the requirement of memory-intensive workloads [10].

We support *RemoteStore* by defining a new store object in *UMap*. *RemoteStore* consists of client-side and provider-side components. The client-side component resides in the application process running on slim nodes. It records information of the remote memory nodes and mapped remote data stores. The provider-side component resides on one or

TABLE 1: Main Configuration Options in *UMap*

Variable	Description
umap_pagesize	the size of internal Upages in memory regions
umap_fillers	the number of workers to fetch from the data store
umap_evictors	the number of workers to evict and write back pages
umap_high_watermark	the threshold of used Upages to start eviction
umap_low_watermark	the threshold of used Upages to stop eviction
umap_bufsize	the maximum DRAM usage for caching pages

multiple memory nodes. It responds to requests from the client-side components, likely from multiple slim nodes, by initiating the actual data transfer and then notifying the client-side when the data transfer completes. RemoteStore uses remote procedure call (RPC) from the Mercury [5] library for fetching pages over the network. This communication layer supports the most common transport protocols on HPC systems, including RDMA, MPI, and TCP.

5 UMAP APIS

UMap is a $C++$ library using the userfaultfd [8] feature of Linux, an asynchronous page fault management protocol available in production kernels since 4.3. *UMap* uses the userfaultfd system call to offload page fault handling in selected virtual memory address into userspace. It uses `UFFDIO_COPY` to atomically copy data to free frames in physical memory and wake up the blocked process. *UMap* can use any underlying file systems, such as XFS and EXT4, for a file-backed data store. It also can use any communication libraries to move pages over the network. *UMap* bypasses the Linux page cache and uses madvise with MADV_DONTNEED to explicitly evict pages from memory. *UMap* is highly multi-threaded: groups of workers at each stage are implemented by POSIX threads. *UMap* defines an abstract store class with common interfaces, such as *r*ead_from_store() and *w*rite_from_store() to support extensibility of data stores. A new type of data store needs to implement the actual data access functions. Each memory-mapped region is backed by a data store whose data access functions are called to resolve page faults in the region.

UMap provides a similar interface to `mmap` to ease porting existing applications. An application registers memory regions to be managed by *UMap* through the `umap` and `uunmap` interface. *UMap* additionally supports a rich set of configuration options at the application level through APIs and environmental variables, e.g., the number of fillers and evictors; the maximum size of physical memory used for buffering pages; and I/O granularities. We list important environmental variables in Table 1.

We demonstrate a simple example in Listing 1. Two types of data stores are memory-mapped into the process's virtual address space. By default, umap() assumes a file-backed data store. Or, as shown in line 7, a custom SparseStore is defined and passed into umap_ex() for memory-mapping. The main computation in the application accesses these memory regions as if to malloc-ed allocations. Finally, resources are released in uunmap().

6 EVALUATION

We evaluate the performance of *UMap* in this section.

Listing 1: An example of memory-mapping two data stores

```
1  //Memory-map a file-backed datastore
2  int fd = open(file_name, O_RDWR);
3  void* addr0 = umap(NULL, region_size1, prot, flags,
       fd, 0);
4
5  //Memory-map a SparseStore as the backing store
6  SparseStore* sstore = new SparseStore(region_size2,
       root_path, per_file_size);
7  void* addr1= umap_ex(NULL, region_size2, prot, flags,
       -1, 0, sstore);
8
9  //Perform the main computation loop
10 compute();
11
12 //Unmap from the virtual address
13 uunmap(addr0, region_size1);
14 uunmap(addr1, region_size2);
```

TABLE 2: The summary of workloads

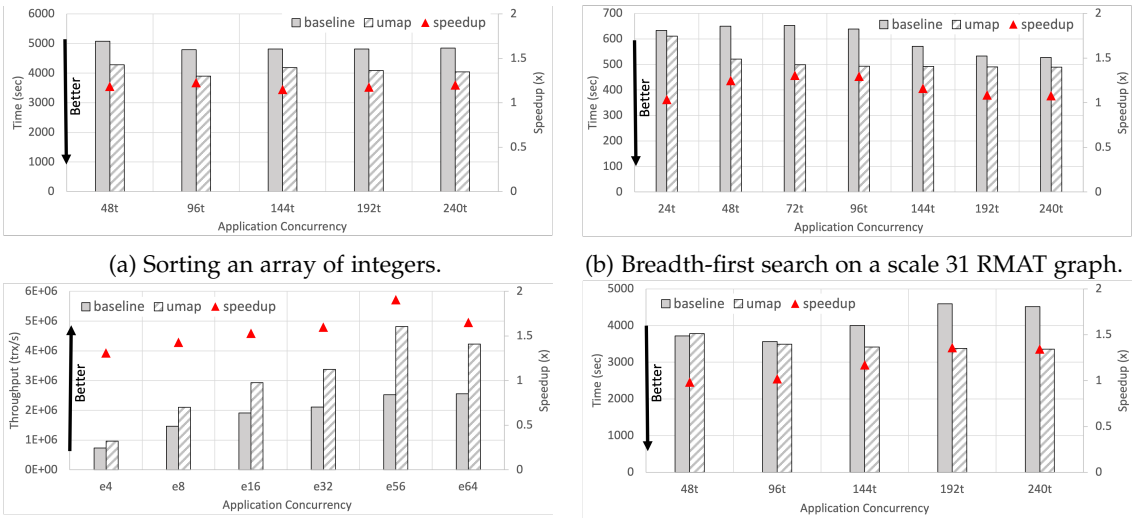
Acronym	Application	Input Size
Sort	data processing - sorting	a 500 GB input array of 64-bit words
BFS	graph processing - breadth-first search	RMAT scale 28-31 graphs (67-530 GB)
NStore	in-memory database	a 384 GB memory pool
LMAT	a metagenomic classification software	a 480 GB k-mer database

6.1 Experimental Setup

Platform. We used AMD testbeds that feature AMD EPYC 7401 (24 cores /48 hardware threads) processors running at 1.2 GHz. The testbed has a total of 256 GB DDR4 DRAM and 3 TB NVMe (type: HGST SN200) SSD mount through the xfs file system. The node architecture is similar to the Exascale Frontier node. The platform runs Fedora 29 with experimental Linux kernels 5.6.0 and 5.1.0[2]. We also use a production cluster called *flash* with Intel processors. Each node has two Intel Xeon E5-2670 v3 processors with 12 cores (24 hardware threads) running at 2.3 GHz (Turbo 3.1 GHz). The machine has 256 GB DDR4 DRAM and 1.5 TB local NVMe SSD (type: HGST SN150) mount through the xfs filesystem. The platform runs the Linux mainline Red Hat Enterprise Linux 7.6 kernel 3.10.0[3].

Workloads. To reflect the data-intensive nature of emerging HPC workloads, we use four applications: out-of-core sort, scale-free graph traversal, database operations, and metagenomic analysis. Table 2 summarizes the applications and their workloads in our evaluation. The applications are compiled using GCC 7.1.0 with OpenMP support. We repeat measurement ten times and report the average execution time or application-defined figure of metric (FoM). Sort is a multi-threaded program that performs quicksort on values stored in a file. The BFS graph traversal benchmark is derived from Graph500 [11]. It performs a level-synchronous breadth-first search (BFS) on an input graph from a source node. We use a Kronecker generator to generate scale-free input graphs saved in CSR (compressed-sparse row) format. The graphs have up to 2.1 billion vertices and 34 billion edges. N-Store [12] is an in-memory database, modified to use *UMap* API by changing approximately ten lines of code. N-Store uses persistent memory as its memory pool for data. N-Store supports multiple executors to execute transactions to the database concurrently. Our evaluation uses the popular YCSB [13] benchmark with eight million transactions

2. The kernel is configured with transparent huge page (THP) always enabled so that khugepaged runs in the background.
3. The kernel is configured with THP disabled.



(a) Sorting an array of integers.

(b) Breadth-first search on a scale 31 RMAT graph.

(c) YCSB transactions on the NStore database.

(d) Metagenomic queries on a k-mer database.

Fig. 3: The overall performance of four data-intensive applications atop *UMap* compared to the baseline at increased application concurrency. All experiments are out-of-memory executions.

and five million keys. The Livermore Metagenomics Toolkit (LMAT [14]) is a production application widely used for scalable genome classification in a metagenomic sample. K-mer queries search a large custom database that typically exceeds the DRAM capacity on a single machine. The lookup process is highly concurrent such that millions of queries are distributed among a group of LMAT threads.

Configurations of *UMap* The rich set of options in *UMap* enables an extensive tuning space. We employ a decision tree to configure critical options based on the application characteristics, instead of sweeping all combinations. The first factor to consider is data locality, e.g., data reuse in fetched pages, either through prior application knowledge or the built-in profiling capability in *UMap*. In general, applications with low data reuse are configured with small page sizes and vice versa. Next, large page sizes need to avoid excessive I/O that bring too much unused data into memory, which is often indicated by performance degradation at increased page size. For applications configured with small pages, further tuning in the high watermark and buffer sizes may increase hits in memory. Finally, the read-write-ratio and the underlying device characteristic can be used to guide the adjustment of the number of fillers and evictors. The extent of tuning efforts is a choice by the user, and autotuning is a possible way to eliminate human efforts.

6.2 Overall Performance

We show that *UMap* improves application scalability over the baseline that uses the current system service (*mmap*) in the kernel in four data-intensive applications at increased application-level concurrency. *UMap* outperforms the baseline at different levels of concurrency (Figure 3), achieving $1.22\times$, $1.30\times$, $1.90\times$, and $1.36\times$ speedup at the highest concurrency, respectively. In this set of experiments, we choose large input problems so that the applications exhibit out-of-core execution that extends their memory with external storage. The data stores in these applications are 150%-200% that of the memory capacity on the testbed. Figure 3 reports the performance and relative speedup by *UMap*.

Out-of-core Sort benchmark performs quicksort on a large data array, i.e., 500 GB in this experiment. Figure 3a shows that *UMap*-based version outperforms the baseline version by 1.15–$1.22\times$ when running with 48 to 240 application threads. The array is a memory-mapped file stored on the local SSD on the AMD testbed. The access pattern in this benchmark has fairly good sequential and temporal locality. Once a page of the memory-mapped region is fetched in, the page is likely to be traversed by read accesses at a stride of eight bytes (i.e., 64-bit words). Write access would only update a page that is already fetched in the in-memory cache, i.e., always a hit in memory. Such access pattern favors large I/O granularity because having fewer page faults decreases the time spent in servicing page faults, and issuing large I/O transfers increases the bandwidth. For this experiment, *UMap* uses 8MB UPages in the memory-mapped region while the baseline ran with 4KB system pages because THP does not support file-backed memory regions. We also performed tests to show that excessively large UPages would reduce the performance because of amplified writes back to the storage. Due to in-place sorting, accesses to the memory-mapped region are read and write mixed. Hence, *UMap* needs to flush a dirty UPage into storage when evicting it from DRAM. Even if only one data element is updated in an evicted page, the whole page needs to be written back to the storage. A possible optimization to reduce write amplification is to keep track of dirtiness in sub-UPage. This experiment also shows that the optimal I/O granularity in an application is often different from a fixed set of configurations supported by the system solution, which requires configurability in the userspace.

Scale-free Graph Traversal is fundamental for graph processing that plays a vital role in social networks, data mining, and bioinformatics. Figure 3b shows that *UMap*-based implementation of BFS outperforms the system service by up to $1.30\times$ on a scale 31 RMAT graph on the

flash testbed. In graph applications, the data structures of graph edges and vertices often dominate the memory footprint, and can easily exceed the memory capacity. In this benchmark, graph structures including edges and vertices are backed by a file-type data stores on the local storage, and memory-mapped into the virtual address space. The input graph has about 2.1 billion vertices with an average degree of 16. The memory-mapped region reached 530 GB so that the execution is out-of-core. BFS is a read-intensive workload because the edge and vertices structure remain unchanged after BFS. Hence, *UMap* does not need to write back to storage when evicting pages from the in-memory buffer. As a level-synchronous implementation of BFS with input graphs in CSR format, the access pattern in this workload has mixed sequential and random accesses, where edges are iterated while vertices may be randomly accessed. The ratio between sequential and random read depends on the input graph, i.e., the distribution of vertex degrees. In general, a graph with more high-degree vertices would have more sequential access. Once a page is fetched in the memory, at least all edges of the vertex will be accessed. A large page size benefits those vertices with high degrees but results in unused data for vertices with low degrees. With refined UPage sizes, *UMap* can better match the structure of an input graph. Figure 3b shows that the system service requires as high as 192 application threads to effectively hide the latency and achieve its best performance. In contrast, *UMap* enables the application to reach optimal performance at any concurrency above 48 threads.

Database Operations. NStore is an in-memory database designed for persistent memory. We ported NStore to use *UMap* APIs by merely changing ten lines of code, where memory-mapped regions are added into its memory pool for data. The experiments used a 384 GB datastore on the local SSD of the AMD testbed. The workload uses the popular YCSB [13] benchmark with eight million transactions and five million keys. These transactions are divided into a group of executors for concurrent execution. In this experiment, we scale up the number of NStore executors from 4 to 64 and report its application-specific throughput in Figure 3c. The access pattern in the benchmark has a relatively low locality because the transactions may access random entries in the database. Therefore, large page sizes would bring unused data, and thus, NStore prefers a relatively small page size of 256KB in contrast to the Sort and BFS benchmarks. We also performed a sensitivity study to confirm that NStore has low sensitivity to the changes in the page sizes as long as they remain smaller than 1MB. The baseline with system service uses the default 4KB page size as it is the page size most close to the access pattern. This workload has mixed read and write accesses so that extra overhead is incurred in *UMap* to ensure updates are persisted into the backend datastores. The speedup by *UMap* compared to the system service scales up at high concurrency at the application level. At the lowest concurrency, NStore by *UMap* achieves $1.30\times$ speedup. When increasing the number of executors, the speedup by *UMap* over the baseline increases substantially to $1.90\times$.

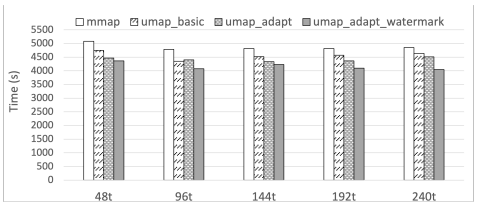
Metagenomic Analysis. LMAT is a production software used for scalable metagenomic classification, including COVID-19-related research. The execution time of taxonomy classification is dominated by calculating the score of taxonomy IDs for each k-mer in a query. The score is calculated by evaluating the list of taxonomy IDs associated with a k-mer in the database. The database is constructed offline from a large collection of reference genome sequence databases and an NCBI taxonomy. As new sequences are added to reference collections, the size of a production k-mer database could reach mulitple TeraBytes in recent development. To support execution on common testbeds with moderate DRAM capacity, the k-mer database is stored in a file and mapped into the LMAT process's virtual address space during taxonomy classification. In this experiment, we use a realistic query set that consists of over 18 million reads. The lookup process is highly concurrent such that millions of queries are distributed among a group of LMAT threads. Each read consists of a set of k-mers. For each k-mer, LMAT needs to query the k-mer database. The lookups in the database result in read-only accesses that are mostly random because k-mers in one read often have little locality in the database. Once a page is fetched into the in-memory buffer, the reuse on the page is also low because only the bytes of k-mers and taxonomy IDs are accessed. Therefore, LMAT benefits from small page size, and both *UMap* and the system service use 4KB page size in this experiment. Figure 3d reports the execution time of the query set at an increased number of LMAT threads on the flash testbed. The results show that *UMap* outperforms the system service when more than 48 LMAT threads are used. The speedup by *UMap* increases when more LMAT threads are used and reached $1.36\times$ at 240 threads. The system service failed to scale at higher concurrency in LMAT, i.e, the execution time increased when more than 144 LMAT threads are used. In contrast, *UMap* sustained the performance of LMAT even at high concurrency.

TABLE 3: The number of major and minor page faults in LMAT with mmap and *UMap* in four production query files.

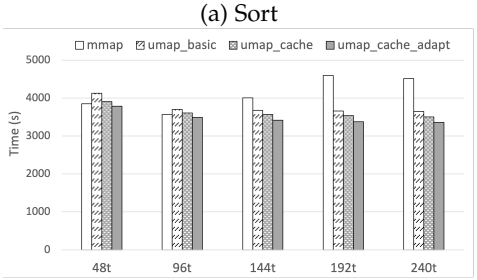
	mmap		*UMap*	
Input Problem	**Major Page Faults**	**Minor Page Faults**	**Major Page Faults**	**Minor Page Faults**
S7	$8.14E+07$	$6.44E+08$	$6.23E+07$	$3.29E+08$
S8	$5.93E+07$	$1.30E+09$	$4.02E+07$	$3.58E+08$
S9	$2.17E+08$	$2.79E+09$	$2.97E+08$	$1.45E+09$
S10	$7.40E+07$	$9.21E+08$	$6.77E+07$	$3.98E+08$

6.3 Performance Analysis

To quantify the reduced kernel overhead in page faults, we measure the number of major and minor page faults in the baseline LMAT with *mmap* and the optimized version with *UMap*. Table 3 reports the comparison in four production inputs. *UMap* reduces the number of major page faults in input S7, S8, and S10 by $1.31\times$, $1.48\times$, and $1.09\times$, and the number of minor page faults by $1.96\times$, $3.62\times$, and $2.32\times$, respectively. For input S9, *UMap* increased the number of major page faults by $1.36\times$ but reduced minor page faults by $1.93\times$. Both types of page faults incur the overhead of fault handling. However, major page faults require extra overhead for I/O to access devices to fetch pages into memory. Thus, comparing the major faults in mmap and *UMap* provides an estimation in data movement from storage to memory. The overall application performance is a mutual result of both page faults.

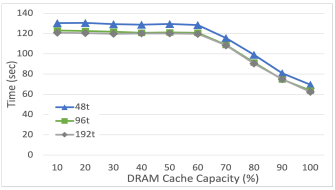


(a) Sort

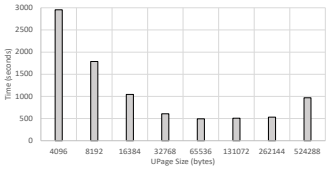


(b) LMAT

Fig. 4: The run time using different optimizations in *UMap*.



(a) Configuration of buffer size.



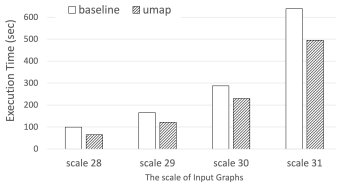
(b) Configuration of page size.

Fig. 5: The impact of different *UMap* configurations on BFS.

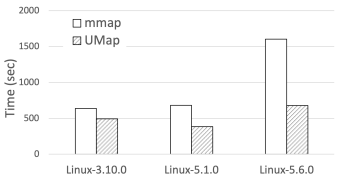
We study the effectiveness of various optimizations in *UMap*. The basic implementation of *UMap* is denoted as *umap_basic*. *UMap* with the dynamic adaptation of memory usage at different levels of concurrency is denoted as *umap_adapt*. The optimization that uses a hybrid of static and dynamic caching in DRAM is denoted as *umap_cache*. *umap_watermark* denotes a high watermark that activates page eviction when 99% of UPages are assigned and deactivates eviction once fewer than 95% of UPages are assigned. For comparison, the execution time of mmap is also presented at each concurrency. Figure 4b shows that LMAT benefits almost equally from the static caching and dynamic memory usage at different concurrencies. LMAT shows little changes to watermarks and thus is omitted in the plot. Unlike LMAT, the Sort application benefits the most from the increased watermark at high concurrency. Figure 4a shows that at 240 application threads, *umap_watermark* brings an additional 1.11× improvement. *umap_cache* is omitted in Sort as it is not applicable based on its access pattern. The results show that no single optimization technique can uniformly benefit all applications and even for the same application, the effectiveness of each optimization may vary as concurrency changes.

6.4 Sensitivity Study

We perform a sensitivity study to understand the impact of *UMap* configurations (i.e., page sizes and buffer sizes), input data, and kernel versions.



(a) Input data size.



(b) Linux kernel versions.

Fig. 6: The impact of input data sizes and kernel versions on BFS.

DRAM Cache Configuration. We configure the buffer size in *UMap* to 10-100% that of the memory-mapped data store. Equivalently, the dataset is 1000-100% memory capacity. Figure 5 reports the execution time of BFS on a scale 29 graph at these configurations. Note that 100% DRAM cache represents the optimal performance. The results show that increased memory capacity may not always improve performance because the performance only starts increasing when more than 60% data can be cached. One takeaway is that if the memory capacity is well below these threshold values, the DRAM resources can be prioritized for other activities.

Page Size Configuration. We evaluate the impact of different configurations of page sizes in Figure 5b. As described in Section 6.2, the BFS benchmark has relatively good locality and favors large page sizes. When increasing the page size from 4KB to 32KB, the execution speeds up steadily six times. From 32KB to 256KB page size, the performance remains stable. However, excessive I/O starts to offset the performance benefit for page sizes at 512KB. Tuning the configuration of page sizes needs to consider both locality and data usage.

Input Data Size. We perform BFS on increased scales of graphs to understand the impact of input data size. All the executions are out-of-core with DRAM usage restricted to 50% the size of each input. Figure 6a shows that *UMap* always outperforms mmap. The speedup by *UMap* over the system service increases when the scale of the input graph increases from 28 to 31, i.e., 268 million to 2.1 billion vertices. When moving towards Exascale, data sets used in applications are expected to increase exponentially [1]. While performance using the standard system service diminishes for the scale of these data-intensive tasks, we demonstrate that a custom service designed for HPC applications can support high performance for large-scale data sets.

Kernel Dependency. *UMap* has a high dependency on the underlying kernels because it is completely implemented inside the userspace. We perform BFS on a scale 31 graph on three kernel versions, i.e., Linux-3.10.0 on the flash node, Linux-5.1.0 and 5.6.0 on two AMD nodes. Note that their processors run at different speeds. For each kernel version, we compare the relative performance between *UMap* and mmap in Figure 6b. As expected, both *UMap* and mmap show high sensitivity to the kernel version. More importantly, Figure 6b shows that *UMap* outperforms the mmap system service on all these kernel versions. This result demonstrates that a userspace service like *UMap* could potentially reduce dependence on kernels on exascale machines and improve performance portability.

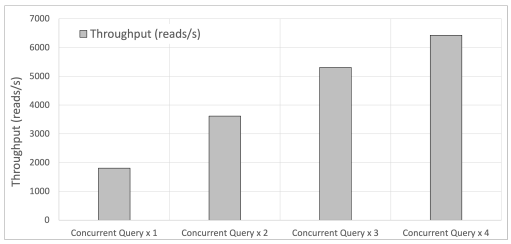


Fig. 7: The aggregate throughput in LMAT when an increased number of client nodes lookup queries in a k-mer database backed by a RemoteStore in *UMap*.

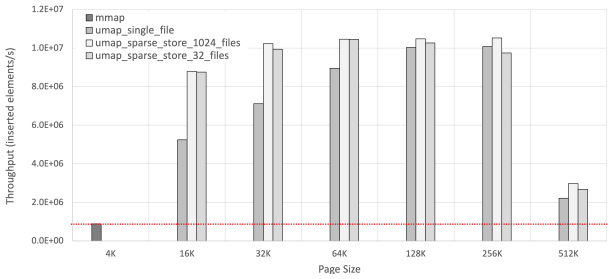


Fig. 8: The throughput of dynamic graph construction using Metall with UMap and two configurations of SparseStore in comparison to mmap (red dotted line).

7 Case Study

We demonstrate two use cases of *UMap* in utilizing RemoteStore for querying the LMAT k-mer database on remote memory and SparseStore in a scalable persistent memory allocator called Metall.

7.1 Case Study I: Database on Remote Memory

In this case study with LMAT, the k-mer database backed by a RemoteStore stays in the physical memory of remote nodes representative of fat memory servers that may become available in heterogeneous sub-clusters on future HPC machines. The porting from the original file-backed data store to a RemoteStore is minimal as the interfaces exposed to the applications are similar (see Listing 1). Transparent to the application, a RemoteStore can be distributed over multiple memory nodes for high aggregated bandwidth on the network. Figure 7 reports the aggregate throughput when running LMAT on one to four client compute nodes. A common k-mer database is distributed across three memory servers. The database, backed by a RemoteStore, is shared by these clients by mapping it into their virtual address space. Each client processes a different set of realistic queries that include 12 million to 49 million reads. Essentially, without porting LMAT into a new programming model, *UMap* transforms a single-node execution model into multi-node execution. The DRAM usage on each client node is limited to 64 GB to emulate slim nodes with small memory capacity on future machines. The scaling results show that the aggregated throughput scales almost linearly as the number of client nodes increases. Note that RemoteStore is implemented for fast prototyping but not optimized for performance. Without the RemoteStore, LMAT must run on a fat memory node and read in the entire database for each run, or must have access to the k-mer database through the file system (node local, rack local, Lustre, . . .) A single RemoteStore flexibly mapped onto a pool of remote memory enables the execution of multi-node queries with a minimal requirement on storage and memory resources.

7.2 Case Study II: A Persistent Memory Allocator

Metall [15] is a C++ memory allocator for persistent memory. Metall was designed to allow data analytics applications to create and access persistent data structures beyond the memory capacity of a single machine. While Metall is a general purpose allocator, in this study we focus on using it for graph applications. Graph analytics applications usually perform data ingestion, which indexes and partitions data with analytics-specific data structures. It is often the case that creating the graph is more expensive than performing any single analytic. Metall reduces the overhead of graph construction by providing a persistent heap to create and update arbitrary dynamically instantiated binary data structures. Metall incorporates the rich C++ interface developed by the *Boost.Interprocess* library to allow applications to store complex custom C++ data structures in persistent memory directly. Internally, Metall is implemented based on memory-mapped files to provide applications with transparent access to data allocated in persistent memory.

We use *UMap* to replace the default system service for memory-mapping files to enable application-specific optimizations in graph algorithms that use Metall. The porting merely requires replacing the use of system `mmap()` with *UMap* APIs. Real-world graphs, e.g., social networks, could have very sparse structures. To optimize storing and processing sparse data structures, we configure Metall to uses the *SparseStore* in *UMap*.

We evaluated the performance using a graph construction benchmark [15]. The benchmark incrementally adds edges to a graph data structure in adjacency list form in a Metall persistent heap. Experiments were performed on the AMD testbed described in section 6.1, generating a graph of size 256 GB and storing it on an NVMe SSD. Note that in intermediate steps there is much higher memory consumption in the persistent heap than the final graph file size. The baseline uses mmap and is denoted as *Metall*+`mmap`. For the *SparseStore* configuration, we further compared the performance of two file granularities, 8192 MBs (32 files), and 256 MBs (1024 files). Figure 8 presents the throughput of constructing a graph in terms of inserted edges per second for three versions of Metall across seven UPage sizes.

Figure 8 shows that Metall with *UMap* outperform the baseline *Metall*+`mmap` by five to 12 times when configured with 16-256K UPages. Increased UPages size improves performance because it reduces the number of page faults and increases the bandwidth utilization by transferring larger chunks of data. We observe performance degradation at the 512KB page size. This could be due to over-fetching unused data into the physical memory. Specifically, the graph construction benchmark's benefit from the largest UPage size is limited due to the benchmark's irregular memory access pattern.

We show that using SparseStore brings an additional

1.17–1.70× improvement compared to the default single-file data store in *UMap* for 16, 32, 64K size UPages. For instance, at 16K UPages, the throughput of graph construction using *Metall+UMap+SparseStore* with 1024 files is 1.7× faster than the default *Metall+UMap*. At increased UPage sizes, the performance benefit from *Metall+UMap+SparseStore* is diminishing, and it reaches almost similar performance at the 256K UPage. One reason for the performance advantage of *SparseStore* for smaller page sizes is the large number of small and random I/Os, which tend to increase contention in the case of a single file [6].

8 Related Work

Kernel-based Approaches DI-MMAP [16] supports scalable memory-mapped I/O on fast storage by combining a loadable kernel module with a runtime to improve page eviction and TLB performance. Several modifications to the Linux kernel are proposed in [17] based on their analysis of the memory management overhead in the Linux virtual memory subsystem for handling memory-mapped I/O. They conclude that kernel-based paging will prevent applications to exploit fast storage. FastMap [6] identifies contention on tree_lock as the main bottleneck for scalability in mmap and proposes per-file and per-core data structures in the kernel for optimizing memory-mapped I/O for fast storage devices. Several filesystems, e.g., NOVA [18] and SplitFS [19], are proposed for fast storage and persistent-memory. Also targeting persistent memory, Kuco [20] combines tasks in the kernel with offloaded tasks in userspace to improve the scalability of filesystems. These filesystems can be leveraged in *UMap* for accessing file-backed data stores. However, making them available on the HPC system is not possible for most HPC end-users, using a facility-defined standard system software installation. Our approach aims to enable high-scalability with the maximum configuration flexibility in userspace. However, a userspace service has limited information and higher overhead compared to the kernel, which is a design tradeoff.

Remote DataStores Many projects have explored remote memory as a way to extend memory capacity based on need. Infiniswap [21] provides a kernel extension to enable transparent remote paging without application modifications. Hailstorm [22] supports a network-attached database deployed on a storage pool. These approaches require kernel modifications and specific hardware. Our extension of RemoteStore in *UMap* provides a way for fast prototyping system changes with minimum dependence on hardware and kernel.

User-level services have been proposed for file systems and paging services. UnifyFS [23] provides a user-level burst buffer file system based on node-local large capacity persistent memory. Co-Pager [24] also provides a user-space paging service with support from a kernel module to reduce the overhead of accessing NVM. Our approach is completely implemented in user-space without any kernel modules and we target to support a variety of backing storage, including NVM and others.

Caching and Data placement are essential for optimizing the performance of memory mapping. Datastores are often placed in a slower tier in the memory hierarchy while good performance requires high hit rates on the fast tier, i.e., the in-DRAM cache. Data placement has been extensively studied for heterogeneous memory systems composed of fast and slow tiers. Xmem [25] characterizes pages based on three access patterns They optimize the placement of pages into fast and slow tiers of a memory system based on their predicted performance gains. Cache partitioning [26], [27], [28], [29] is often proposed for processor's shared caches to avoid the interference in multi-program workloads that exhibit different memory access patterns. [29] reconfigure the cache partition using reuse distance to predict cache miss rate at different cache partitions. Many of these optimizations can be supported through configurability in the caching policies in *UMap*.

9 Conclusions

In this work, we prioritize scalability and extensibility as the main design principles to support memory mapping in userspace for applications on exascale compute nodes. We present *UMap*, a userspace memory mapping service that requires no modifications to the kernel or new kernel modules to support fine-grained configuration control. New types of data stores can be easily extended in *UMap* through the *DataStore* abstraction. With decoupled queue management, concurrency-aware adaptation, and dynamic load balancing, *UMap* supports applications to scale even at the high CPU concurrency of exascale nodes. We evaluate *UMap* in data-intensive applications, including a production metagenomic classification code. Results show that *UMap* can outperform Linux's mmap at various levels of concurrency. In two case studies, we demonstrate the extensibility of *UMap* in supporting a new datastore on remote memory over the network and a SparseStore for use in a persistent memory allocator.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-JRNL-819817). This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

[1] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, 2015.

[2] M. Asch, T. Moore, R. Badia, M. Beck, P. Beckman, T. Bidot, F. Bodin, F. Cappello, A. Choudhary, B. de Supinski *et al.*, "Big data and extreme-scale computing: Pathways to convergence-toward a shaping strategy for a future software and data ecosystem for scientific inquiry," *The International Journal of High Performance Computing Applications*, vol. 32, no. 4, pp. 435–479, 2018.

[3] I. Peng, K. Wu, J. Ren, D. Li, and M. Gokhale, "Demystifying the performance of hpc scientific applications on nvm-based memory systems," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 916–925.

[4] J. Lofstead, "Memory vs. storage software and hardware: The shifting landscape," in *Smoky Mountains Computational Sciences and Engineering Conference*. Springer, 2020, pp. 394–407.

[5] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham *et al.*, "Mochi: Composing data services for high-performance computing environments," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 121–144, 2020.

[6] A. Papagiannis, G. Xanthakis, G. Saloustros, M. Marazakis, and A. Bilas, "Optimizing memory-mapped I/O for fast storage devices," in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020, pp. 813–827.

[7] I. Peng, M. McFadden, E. Green, K. Iwabuchi, K. Wu, D. Li, R. Pearce, and M. Gokhale, "Umap: Enabling application-driven optimizations for page management," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 2019, pp. 71–78.

[8] Linux, "Userfaultfd," https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html, 2021.

[9] I. B. Peng, M. B. Gokhale, and E. W. Green, "System evaluation of the intel optane byte-addressable NVM," in *Proceedings of the International Symposium on Memory Systems*. ACM, 2019.

[10] I. Peng, R. Pearce, and M. Gokhale, "On the memory underutilization: Exploring disaggregated memory on hpc systems," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2020, pp. 183–190.

[11] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[12] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 707–722.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[14] S. K. Ames, D. A. Hysom, S. N. Gardner, G. S. Lloyd, M. B. Gokhale, and J. E. Allen, "Scalable metagenomic taxonomy classification using a reference genome database," *Bioinformatics*, vol. 29, no. 18, pp. 2253–2260, 2013.

[15] K. Iwabuchi, L. Lebanoff, M. Gokhale, and R. Pearce, "Metall: a persistent memory allocator enabling graph processing," in *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 2019, pp. 39–44.

[16] B. Van Essen, H. Hsieh, S. Ames, R. Pearce, and M. Gokhale, "DI-MMAP–a scalable memory-map runtime for out-of-core data-intensive applications," *Cluster Computing*, 2013.

[17] N. Y. Song, Y. Son, H. Han, and H. Y. Yeom, "Efficient memory-mapped I/O on fast storage device," *ACM Transactions on Storage (TOS)*, vol. 12, no. 4, p. 19, 2016.

[18] J. Xu and S. Swanson, "{NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, 2016, pp. 323–338.

[19] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "Splitfs: Reducing software overhead in file systems for persistent memory," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 494–508.

[20] Y. Chen, Y. Lu, B. Zhu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Shu, "Scalable persistent memory file system with kernel-userspace collaboration," in *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, 2021.

[21] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 649–667.

[22] L. Bindschaedler, A. Goel, and W. Zwaenepoel, "Hailstorm: Disaggregated compute and storage for distributed lsm-based databases," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 301–316.

[23] A. Moody, D. Sikich, N. Bass, M. J. Brim, C. Stanavige, H. Sim, J. Moore, T. Hutter, S. Boehm, and K. Mohror, "Unifyfs: A distributed burst buffer file system-0.1. 0," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2017.

[24] F. Li, D. G. Waddington, and F. Song, "Userland co-pager: boosting data-intensive applications with non-volatile memory, userspace paging," in *Proceedings of the 3rd International Conference on High Performance Compilation, Computing and Communications*. ACM, 2019, pp. 78–83.

[25] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 15.

[26] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *The Journal of Supercomputing*, vol. 28, no. 1, pp. 7–26, 2004.

[27] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 423–432.

[28] Y. Oh, J. Choi, D. Lee, and S. H. Noh, "Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems." in *FAST*, vol. 12, 2012.

[29] P. Li, C. Pronovost, W. Wilson, B. Tait, J. Zhou, C. Ding, and J. Criswell, "Beating opt with statistical clairvoyance and variable size caching," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 243–256.

Ivy B. Peng (Member, IEEE) is a Computer Scientist in the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory, USA. Dr. Peng received her Ph.D. degree in Computer Science from KTH Royal Institute of Technology, Sweden. Her research interests revolve around high-performance computing with focuses on emerging heterogeneous architecture, memory systems, performance characterization, and optimizations.

Maya Gokhale (Fellow, IEEE) is a Distinguished Member of Technical Staff with the Lawrence Livermore National Laboratory, USA. She received a Ph.D. degree in Computer Science from University of Pennsylvania. She was the co-recipient of an R&D100 award for a C-to-FPGA compiler, co-recipient of four patents, and co-author of more than one hundred technical publications. She is on the Editorial Board of the Proceedings of the IEEE and Associate Editor of IEEE Micro. Her research interests include data-intensive architectures and reconfigurable computing.

Karim Youssef received BS in Computer and Communication Engineering from Alexandria University, Egypt. He is currently a Ph.D. student with the Department of Computer Science at Virginia Tech, Blacksburg, Virginia. His research interests include scaling the performance of big-data analytics applications using high-performance computing technologies.

Keita Iwabuchi is a data scientist in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His research area is distributed systems and parallel computing, particularly in high-performance computing. Major focuses are HPC-scale graph analytics and system software for persistent memory and non-volatile memory. He received his Ph.D. in Mathematical and Computing Sciences from Tokyo Institute of Technology.

Roger Pearce received a Ph.D. in Computer Science form Texas A&M University in 2013, and a B.S. in Computer Engineering in 2004. He is a computer scientist in the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory. His research interests center around parallel and external memory graph algorithms and data-intensive computing on HPC systems.