Privateer: Multi-versioned Memory-mapped Data Stores for High-Performance Data Science

Karim Youssef*†, Keita Iwabuchi†, Wu-chun Feng*, and Roger Pearce†
*Department of Computer Science, Virginia Tech
{karimy,wfeng}@vt.edu
†Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
{iwabuchi1,rpearce}@llnl.gov

2021 IEEE High Performance Extreme Computing Conference (HPEC) | 978-1-6654-2369-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/HPEC49654.2021.9622826

***Abstract*—The exponential growth in dataset sizes necessitates the use of high-performance computing (HPC) for large-scale data science. Furthermore, the sizes of these datasets shift the performance bottleneck from the compute subsystem towards the memory and I/O subsystems. To address this shift, modern HPC clusters are equipped with low-latency and high-bandwidth storage devices, such as non-volatile memory, and, in turn, re-designed I/O subsystems to improve performance. However, an overlooked bottleneck arises due to the size of these storage devices being dwarfed by the storage footprint of large-scale data science applications. For instance, applications that process and store consistent snapshots of incrementally growing data streams require a significant storage footprint that far outstrips the size of the storage devices.**

To address this bottleneck, we present Privateer, a general-purpose data store that optimizes the tradeoff between storage space utilization and I/O performance. Privateer uses memory-mapped I/O with private mapping and an optimized writeback mechanism to maximize write parallelism and eliminate redundant writes; it also uses content-addressable storage to optimize storage space via de-duplication. We evaluate the effectiveness of Privateer by using it as the data-store management layer of Metall, a persistent C++ data structure allocator. Using a micro-benchmark that incrementally constructs and stores snapshots of an incremental graph data structure, Privateer can reduce the storage space used by approximately 30% while delivering comparable performance to the baseline Metall implementation.

I. Introduction

The exponential growth in dataset sizes represents a challenge as well as an opportunity for data science to extract knowledge and insight from this wealth of data. This growth, along with the increasing computational complexity of data-science algorithms, motivates the need for high-performance computing (HPC) systems [1]. However, this growth has caused the performance bottleneck to shift from the compute subsystem to the memory and I/O subsystems. To address this shift, modern HPC systems incorporate fast storage technology, such as non-volatile memory [2], whose low latency and high bandwidth offer multiple opportunities to enhance the I/O performance of large-scale data science applications [3].

Past research has focused on maximizing the I/O performance of data-science applications via an optimized I/O subsystem. Such efforts include re-designing the memory-mapped I/O path in Linux systems [3] and enabling application-specific virtual memory management (VMM) in user space [2]. Other efforts include providing productive data-store interfaces built on top of optimized memory-mapped I/O [4] and persistent data-structure allocation [5], [6]. While these efforts offer promising solutions by leveraging the performance of fast storage devices and by managing data stores using productive interfaces, they overlook the bottleneck of limited capacity in these storage devices, relative to the storage needs of data-science applications [7], particularly those that need to store multiple snapshots of evolving data stores [8].

As shown in [7], simultaneously optimizing storage space utilization and I/O performance is a challenge. For instance, optimizing storage space via compression incurs extra CPU overhead to compress and decompress data. Existing solutions that optimize *both* parameters are tailored for specific types of data stores. For instance, LLAMA optimizes I/O performance and storage space for snapshotting evolving graphs using multi-versioned compressed sparse row (CSR) arrays [9]. On the other hand, RocksDB [7] and Kreon [10] optimize both parameters for key-value stores by using different log-structured merge (LSM) tree [11] compaction algorithms. While these data-structure level optimizations guarantee a near-optimal tradeoff between storage space and I/O performance, their benefit is limited to their respective types of data stores.

To address these challenges, we present Privateer, a general-purpose data store that optimizes I/O performance and storage space at the abstraction level of virtual memory and backing store management. Privateer uses private memory-mapping with optimized writeback and content-addressable data storage; it also supports data-store versioning via an efficient virtual memory snapshot interface. We show the efficacy of Privateer by using it as the storage management layer of Metall, a C++ persistent data structure allocator for non-volatile memory devices [5], [12]. Privateer can deliver 30% storage space improvement for storing snapshots of an incrementally growing graph while delivering comparable performance to the baseline Metall. We summarize our contributions below:

- The design and implementation of Privateer, an open-source data store for fast storage devices that optimizes I/O performance and storage space utilization.[1]
- A performance evaluation of Privateer through a case study of persisting snapshots of an incrementally evolving graph data structure.

[1]The code will be made available at https://github.com/LLNL/Privateer

- An analysis of the tradeoff between storage space optimization and I/O performance.

The rest of the paper is organized as follows. First, §II describes the design of Privateer, and §III provides a description of our experiment setup and evaluation of Privateer. Next, §IV discusses related work; §V discusses future directions; and finally, §VI concludes.

II. Design and Implementation of Privateer

The design goals of Privateer are two-fold: (1) providing a data-store management interface that enables efficient versioning capabilities and supports different types of data stores and (2) optimizing the tradeoff between storage space utilization and I/O performance. Storing multiple snapshots of a data store requires optimizing storage space as well as I/O performance to cope with the exponential growth in dataset sizes [13]. Optimizing these two parameters is a challenging tradeoff [7]. For instance, optimizing storage space via data compression incurs extra CPU overhead for compressing and decompressing data. To address this challenge, Privateer adopts a content-addressable storage (CAS) technique to optimize storage space via de-duplication [14], [15]. CAS optimizes storage space utilization at the cost of extra CPU overhead when writing a data block to compute the block hash [15]. To offset this overhead, Privateer optimizes writeback performance by parallelizing the hash computations and block writes. Moreover, Privateer eliminates redundant writes of unchanged data by querying Linux's *pagemap* [16] information to identify the dirtied virtual memory pages and only writing these pages back, as described later in this section.

Privateer encompasses three main layers: the API layer, the virtual memory management (VMM) layer, and the block storage layer. Figure 1 provides a high-level description of Privateer's architecture.

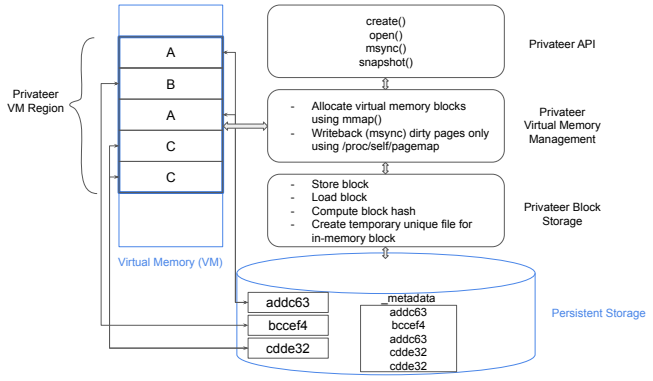


Fig. 1: A high-level overview of Privateer's architecture. Privateer consists of an API layer for end users, virtual memory management (VMM), and a block storage layer that manages the backing store using content-addressable storage (CAS).

A. Privateer's API

Privateer provides APIs for creating a new memory-mapped data store, opening an existing data store, flushing data back to the backing store (i.e., msync), and snapshotting the current in-memory data into the backing store. The *create* API allocates a virtual memory region with a size of *max_capacity* using the *mmap* system call. This region is then transparently partitioned into multiple sub-regions, where a backing file is dynamically created for each sub-region on-demand. A detailed description of the virtual memory management (VMM) technique is described later in this section.

The *open* API opens an existing data store and maps it into the application's virtual memory space. Privateer stores data on the backing store using a content-addressable storage (CAS) method. Privateer's data is stored under two directories. The `blocks` directory stores the data blocks, while the metadata directory stores the file recipe used to reconstruct the region from the data blocks. Other metadata is also stored, such as the current region size and the path to the `blocks` directory.

The *msync* API writes the data back to the backing store. We note that *msync* must be called explicitly by the application to guarantee proper backing-store synchronization.

Finally, the *snapshot* API saves a named snapshot of the current in-memory data store to the backing store. The difference between the *snapshot* and *msync* APIs is that *snapshot* creates a new metadata directory with a specific name passed to the API, while the *msync* API updates the current metadata directory that was specified through the *create* API.

B. Virtual Memory Management (VMM)

The virtual memory management (VMM) layer allocates virtual memory (VM) and backing store space using memory-mapped I/O. Memory-mapped I/O has been shown to provide multiple advantages for applications that process large datasets and leverage state-of-the-art fast storage technology [2], [3]. When a new data store is created, a region with size *max_capacity* is allocated using non-writable anonymous mapping (i.e., without a backing file). This technique allows Privateer to allocate the maximum allowable VM space. After that, the allocated region is transparently partitioned into multiple blocks of size *block_size*, which is set using an environment variable. For each block up to *current_size / block_size*, where *current_size* is provided through the *create* and the *resize* APIs, a new memory mapping is created that overwrites the anonymous mapping for the sub-region constituting the block. The new mapping is writable and uses a private, copy-on-write memory mapping, i.e., MAP_PRIVATE. Using MAP_PRIVATE means that the system's *msync* is not called in the background, as in the case of MAP_SHARED. To write data back to the backing store, Privateer's *msync* uses Linux's *pagemap* [16] information to identify and write only the dirtied pages. To identify a dirty page, Privateer checks two flags in */proc/self/pagemap*, the *present* flag and the *file_page* flag. Since MAP_PRIVATE uses copy-on-write, a dirty page in a private mapping is present but is no longer a file page. This design overcomes multiple limitations of the system's *msync*, including random and bursty I/O at non-deterministic time intervals, as well as other system overheads [3]. We adapted

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

this design from [12] and modified it to work with content-addressable storage of backing files.

A backing file is created for any modified block at the first call to *msync* after the block was modified. This design has multiple advantages. First, the VM region partitioning increases the writeback parallelism as multiple threads can handle multiple blocks simultaneously without synchronization. Second, the on-demand creation of backing files optimizes the physical storage space for sparse data. Finally, it enables further storage optimization using de-duplication of blocks via content-addressable storage, as described later in this section.

C. Block Storage Management

The block storage layer stores data blocks using content-addressable storage (CAS). CAS is a method for optimizing storage space via de-duplication [14]. It works by storing blocks into files with names that uniquely represent the block's content, e.g., SHA-256 hash. This means that duplicate data blocks are stored only once. A metadata file saves the order of the data blocks in the data store. Loading a data store incurs only minimal overhead by scanning the metadata file to read the data blocks in the right order.

A principal design parameter that determines the efficiency of CAS is the block size. Smaller block sizes typically yield more savings in storage space since they result in a higher probability of duplication. This improvement in storage space also depends highly on the characteristics of the dataset. That is, storage saving is inherently inversely proportional to the block size. The disadvantage of using smaller block sizes is that they require more hash computations. Moreover, synchronization is required between Privateer's parallel *msync* threads to avoid data races when writing to a file with the same hash. To prevent these synchronizations from reducing the parallel I/O efficiency, Privateer's block storage layer creates a temporary file with a unique name for each block. This allows the virtual memory management (VMM) layer to perform a parallel *msync* without synchronization between the writing threads. After a block is written into the temporary unique file, the block hash is computed, and the block is either renamed to its content-based hash or removed if another block already holds the same name. This approach requires synchronization only for renaming or removing temporary files. In our implementation, we used the C++ atomic *rename* function, which fails if a file with the same name already exists. Catching this failure with a *file_exists* error code, the temporary file is instead removed. While this approach can result in redundant writes of blocks that have the same content, it increases write parallelism by avoiding explicit synchronization. Our preliminary results show that this approach is nearly five times faster than using a named mutex or a global semaphore. On the other hand, the maximum number of redundant blocks written during a single *msync* operation amounted to only 0.15% of the total number of written blocks. However, the increasing number of file system operations required to rename or remove temporary blocks still affects performance at smaller block sizes (e.g., 128KB). A detailed analysis of the tradeoff between writeback performance and storage savings is provided in §III

III. Evaluation

Our evaluation of Privateer encompasses three goals: (1) the efficacy of Privateer's storage space optimization, (2) the impact of Privateer optimizations on application performance, and (3) the analysis of the tradeoff between storage space optimization and application performance. For our evaluation, we use a micro-benchmark that incrementally constructs a persistent C++ graph data structure using Metall [5].

Metall is a persistent C++ data structure allocator optimized for fast storage devices, such as persistent memory. Metall allows data analytics applications to persist the data structures and re-attach to them for subsequent analysis, hence reducing the cost of raw data ingestion, indexing, and partitioning for each subsequent analysis. Metall also enables versioning of data structures for applications that require storing and processing consistent snapshots of dynamically changing data, such as incremental graphs [8].

We modify Metall's storage manager and memory allocator to use Privateer, as described below. Our micro-benchmark incrementally constructs an adjacency list from a raw graph dataset. It also periodically stores temporal snapshots of the dynamically constructed adjacency list. We use a dump of the Wikipedia page reference graph from 2001 until 2017 to incrementally construct the adjacency list and save monthly and yearly snapshots. Next, we provide a detailed description of the experimental setup, followed by our results.

A. Experimental Setup

1) Metall+Privateer: Metall uses a memory-mapped data store to persist the allocated data structures. We integrate Privateer into Metall by extending Metall's data store manager to use a Privateer object. Like Privateer, Metall's default data store manager allocates a large contiguous virtual memory (VM) region and then dynamically partitions it into multiple chunks. Our extended data store manager uses Privateer's *create()* API to overwrite this memory-mapped region with a Privateer region. We also modify Metall's data store's *open()*, *flush()*, and *snapshot()* to use Privateer APIs. For the experiments in this paper, we use Metall without Privateer as our baseline for comparison. To accurately measure the tradeoff between I/O performance and storage space optimizations, we use a version of Metall that incorporates the *pagemap* writeback optimizations [12].

2) Incremental Graph Construction: We use a micro-benchmark that uses Metall to incrementally construct a persistent adjacency list data structure from a real-world graph dataset. The input to the micro-benchmark consists of an edge list where each item contains a pair of vertex IDs and a timestamp. The micro-benchmark ingests the edge list into an adjacency list data structure that is allocated using Metall. It then saves periodic snapshots of the adjacency list at a configurable time period. In our evaluation, we experiment with monthly and yearly snapshots.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

The baseline Metall implementation provides a snapshot API that copies the entire data store into a snapshot directory. Metall+Privateer only writes new or modified data blocks to Privateer's `blocks` directory and generates a snapshot consisting of a new metadata directory, as described in §II.

3) Dataset: We use a real-world graph dataset that consists of a dump of the Wikipedia page reference graph. The Wikipedia page reference graph consists of all the Wikipedia pages hyperlinks from January 2001 until July 2017. The graph contains a total of 1.8 billion edges (hyperlinks).

4) Computing Platform: We conduct our evaluation on Corona, a cluster that consists of 270 compute nodes. Each node consists of a 48-core AMD EPYC 7401 processor, 256GB DRAM, and 1.6TB NVMe SSD local storage. Corona nodes are equipped with RedHat Enterprise release 7.9 and Linux kernel version 3.10.

B. Results

We first evaluate the effectiveness of Privateer's storage space optimizations. Since Privateer optimizes storage space using de-duplication, a smaller block size is expected to yield more savings with respect to storage space. Hence, we vary the Privateer block size from 256MB down to 128KB. We also characterize the implications of decreasing Privateer's block size on performance by measuring the snapshotting time as well as the total micro-benchmark running time for both the baseline Metall and Metall+Privateer. We then use this evaluation to analyze the tradeoffs between storage space optimizations and I/O performance.

1) Storage Space Optimization: Figure 2 shows the cumulative snapshots size after each yearly (2a) and monthly (2b) snapshot of the Wikipedia page reference graph. Decreasing the Privateer block size yields a consistent improvement in storage space utilization. The total size of all yearly snapshots is 163GB for baseline Metall, compared to 118GB for Metall+Privateer with a 128KB block size, yielding a 27.6% savings in storage space. For the monthly snapshots, baseline Metall's total size of monthly snapshots exceeds the compute node's capacity of SSD local storage ($\sim$1.6 TB). Thus, the benchmark can only store 188 out of the 199 monthly snapshots when using baseline Metall. With Metall+Privateer, the benchmark stores all 199 snapshots for a total size of 1.12TB. (After 188 monthly snapshots, the total snapshots size was 1.46TB for baseline Metall, compared to 0.972TB for Metall+Privateer, yielding a 33.2% savings in storage space.)

2) Runtime Performance: Figure 3 shows the total benchmark execution time for different Privateer block sizes. The total runtime includes the time to ingest the raw edge list into the adjacency list data structure, the time to save the snapshots, and the time to close the Metall manager. Saving snapshots includes writing data back into the backing store and creating a copy of the data store into a separate directory. As described in §II, Privateer optimizes the snapshotting time by copying only the metadata, instead of copying the entire data store. For the yearly snapshotting benchmark, this optimization yields a 3.47$\times$ speedup in snapshotting time and a 1.3$\times$ speedup in total runtime for the 256MB block size, compared to baseline Metall. For the monthly snapshots, the 256MB block size yields a 2.6$\times$ speedup in snapshotting time and a 1.68$\times$ speedup in total runtime. For smaller Privateer block sizes, the speedup gain diminishes as it is offset by the overhead of the file system operations and hash computations necessary for Privateer blocks naming. For the 128KB Privateer block size and the yearly snapshots, the snapshotting time and the total runtime are comparable to baseline Metall. On the other hand, the monthly snapshots incurred a slowdown of 1.1$\times$ in snapshotting time and 1.29$\times$ in total time. These results indicate a tradeoff between storage space optimization and runtime performance.

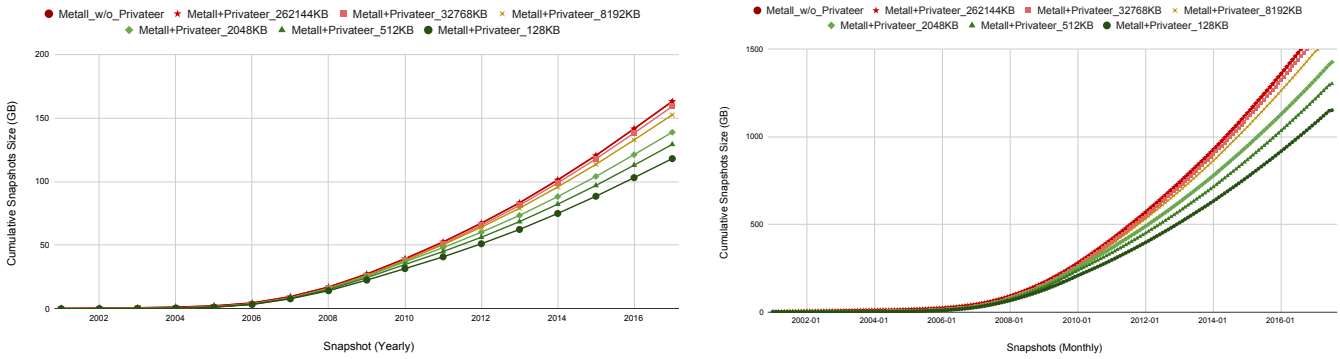
3) Storage Optimization and Performance Tradeoff: Figure 4 presents an analysis of the tradeoff between storage optimization and runtime performance. Larger block sizes (8MB and larger in this case) yield an improvement in runtime due to Privateer's snapshot optimizations. On the other hand, block sizes smaller than 8MB yield a significant and consistent improvement in storage space. This storage optimization comes at the cost of reduced performance gain. As described earlier, a smaller block size implies more file system operations and hash computations. For our use case, the block size that optimizes this tradeoff is 1MB. In general, the optimal point is a function of multiple factors, such as the characteristics of the data as well as the properties of the underlying file system. For instance, a dataset that has more inherent duplicate patterns could benefit from storage optimizations at a larger block size, which would also achieve a performance improvement. On the other hand, a file system with different characteristics, such as a network-based file system, would suffer more performance degradation at smaller block sizes due to remote file system operations. As part of our future work, we seek to analyze all the parameters affecting this tradeoff in order to identify the optimal configuration of Privateer's block size.

IV. Related Work

A. Optimizing I/O Performance Using Memory Mapping

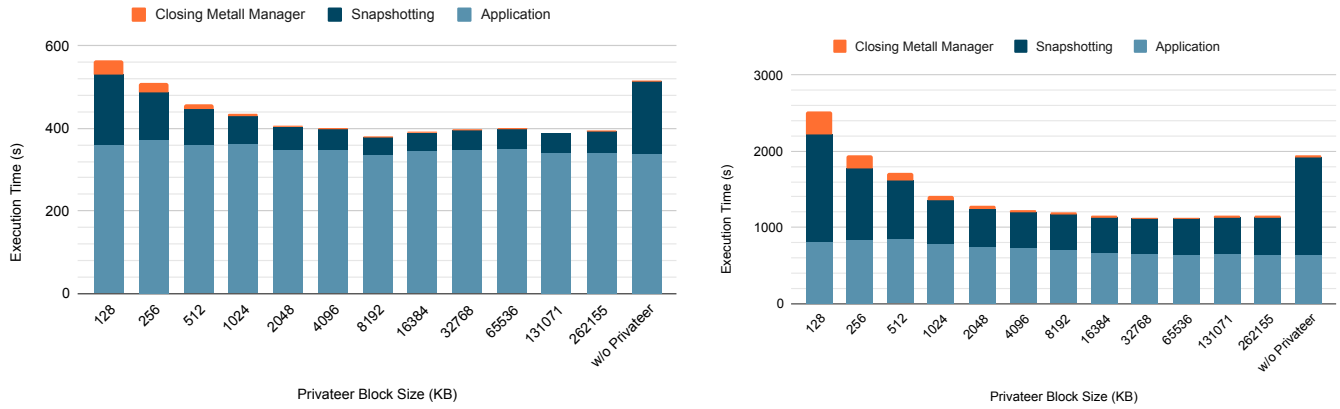
Multiple research efforts [2], [3], [10] have focused on optimizing memory-mapped I/O [17] to leverage its advantage over explicit read-write I/O for fast storage devices. While FastMap [3] optimizes the scalability of Linux's *mmap* for multi-threaded applications, using their optimizations requires a specific kernel module, which affects its portability. On the other hand, UMap [2] allows for application-specific paging optimizations in user space; however, it only works with Linux kernel versions 5+ to support write-protected mappings in user space. We also provide a user-space implementation of *msync()* to optimize both writeback performance and storage space via CAS. As part of future work, we plan to leverage UMap for further I/O optimizations and apply FastMap's optimizations in user space for better portability.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE



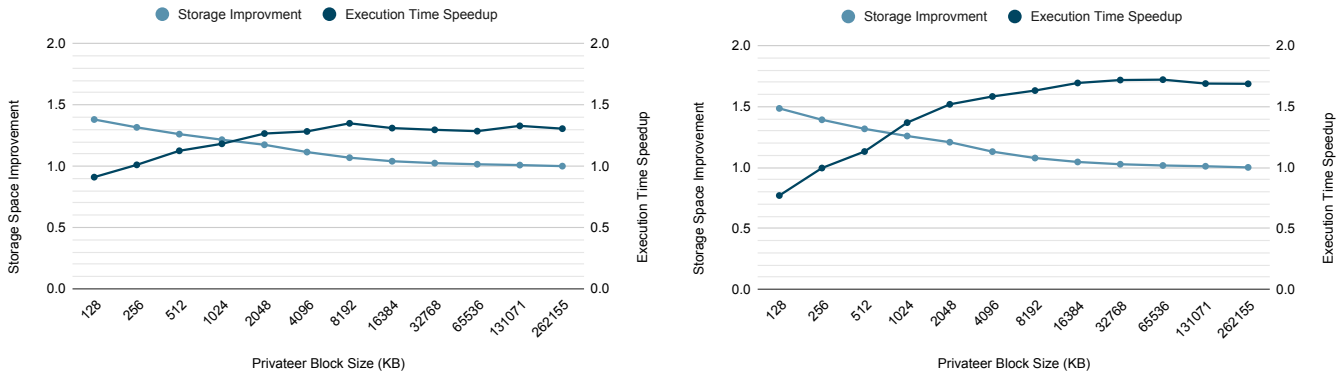
(a) (b)

Fig. 2: Cumulative snapshots size for incremental graph construction and storage of (a) yearly snapshots and (b) monthly snapshots using Metall and the Wikipedia page reference graph. Each line plot represents a different Privateer block size.



(a) (b)

Fig. 3: Effect of varying Privateer's block size on the total running time of the incremental graph construction and storage of (a) yearly snapshots and (b) monthly snapshots. The total time for baseline Metall is shown in the last column for reference.



(a) (b)

Fig. 4: Tradeoff between storage improvement and running time speedup for the incremental graph construction and storage of (a) yearly snapshots and (b) monthly snapshots.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

B. Memory-Mapped Data Stores

UMap provides extensible data store objects that define how data are stored on, read from, and written into the backing store [4]. This allows UMap users to implement storage-specific data store interfaces to further optimize I/O performance and storage efficiency. For instance, SparseStore [18] is a memory-mapped data store interface that partitions the memory-mapped data into multiple blocks to optimize the storage of sparse data and to maximize parallel I/O. A part of Privateer's design was inspired from SparseStore; however, Privateer augments this design concept with content-addressable storage for de-duplication. Privateer also uses regular *mmap()* to be compatible with current production Linux kernels. As mentioned earlier, Privateer could benefit from UMap in the future to enable application-specific optimizations.

C. Optimizing I/O Performance and Storage Space

Optimizing both I/O performance and storage space is a challenging tradeoff as more space optimizations typically imply more CPU time [7], [15]. Existing solutions optimize both parameters for specific types of data stores using data-structure level optimizations.

1) Evolving Snapshots of Graphs: LLAMA optimizes the storage of multiple snapshots of an evolving graph using multi-versioned compressed sparse row (CSR) arrays [9]. The benefits of these optimizations are limited to graph data stores. On the other hand, Privateer optimizes storage utilization and I/O performance at the virtual memory and backing store management level, rendering it applicable to various types of data stores. Moreover, LLAMA could use Privateer as a data store management layer for further I/O and storage optimizations.Other graph data stores include GraphOne [19], a data store that optimizes concurrent ingestion and access of graph data. GraphOne stores snapshots of evolving graphs using an edge list format. Similar to LLAMA, these optimizations are specific to graph data stores.

2) Key-Value Store Databases: RocksDB is a key-value store database that is optimized for flash storage [7]. RockDB optimizes I/O performance and storage efficiency by introducing algorithmic optimizations and compression techniques for persisting a log-structured merge (LSM) tree. Kreon introduces further algorithmic optimizations along with optimizes memory-mapped I/O [10]. These optimizations are specific to key-value stores implemented using variants of an LSM tree. We believe that Privateer could provide additional optimizations to such tools by further optimizing the storage of the memory-mapped data blocks.

V. Future Directions

There are multiple future directions for Privateer to support different applications, architectures, and application-specific optimizations.

A. Privateer for Key-Value Data Stores

Key-value data stores such as RocksDB [7] and Kreon [10] are widely used in the big data industry. Optimizing key-value storage and performance for flash storage devices have attracted significant research attention. As a future direction, we plan to explore using Privateer as a storage optimization layer for key-value data stores with no or minimal changes to the data structure design. Privateer could provide an extra storage optimization layer below the existing data structure optimizations. However, a study of the storage and performance tradeoff needs to be conducted to analyze the benefits and tradeoffs of such design.

B. Privateer and User-Space Paging

A key design parameter for scalable data stores on fast storage devices is I/O performance. As described earlier, memory-mapped I/O provides a considerable opportunity for performance optimization. A current limitation of Privateer is that smaller block sizes result in a significant number of memory mappings that approach the system's limit. Moreover, kernel-space memory-mapped I/O lacks flexibility for application-specific optimizations [2]. As a future direction, we plan to leverage user-space paging, such as UMap, to address these limitations.

C. Distributed Block Storage Service

Privateer's block storage layer leverages content-addressable storage (CAS) to optimize storage space. Other benefits of CAS include network bandwidth optimization for sharing blocks between distributed processes [15]. As a future step, we plan to design and evaluate a distributed block storage service. There are multiple design considerations, such as concurrency control on a distributed file system and optimizing blocks transfer via a local per-node blocks cache.

VI. Conclusion

Privateer is a data-store management interface that optimizes I/O performance and storage space utilization for fast storage devices such as NVRAM. Privateer optimizes writeback performance using private, copy-on-write memory mapping and a user-space *msync* implementation. To optimize storage space, Privateer uses a content-addressable storage model that partitions the virtual memory regions into blocks, creates a backing file for each block, and names blocks using the SHA-256 hash of the block's content. This approach optimizes storage space utilization by storing duplicate blocks only once. Privateer achieves 33% storage improvement for storing multiple snapshots of an evolving graph using Metall, a C++ persistent data structure allocator. Privateer optimizes the tradeoff between I/O performance and storage space using a configurable block size that could be set to satisfy application-specific I/O and storage requirements.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

ACKNOWLEDGMENT

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-CONF-824675). Funding from LLNL LDRD project 21-ERD-020 was used in this work. This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Additionally, we would like to thank Ben Woodard (RedHat) for his informative discussion on Linux VM management.

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] 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.

[3] 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.

[4] I. B. Peng, M. Gokhale, K. Youssef, K. Iwabuchi, and R. Pearce, "Enabling scalable and extensible memory-mapped datastores in userspace," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[5] 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.

[6] W. Cai, H. Wen, H. A. Beadle, C. Kjellqvist, M. Hedayati, and M. L. Scott, "Understanding and optimizing persistent memory allocation," in *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, 2020, pp. 60–73.

[7] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb." in *CIDR*, vol. 3, 2017, p. 3.

[8] J. Gao, C. Zhou, and J. X. Yu, "Toward continuous pattern detection over evolving large graph with snapshot isolation," *The VLDB Journal*, vol. 25, no. 2, pp. 269–290, 2016.

[9] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 363–374.

[10] A. Papagiannis, G. Saloustros, G. Xanthakis, G. Kalaentzis, P. Gonzalez-Ferez, and A. Bilas, "Kreon: An efficient memory-mapped key-value store for flash storage," *ACM Transactions on Storage (TOS)*, vol. 17, no. 1, pp. 1–32, 2021.

[11] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[12] K. Iwabuchi, K. Youssef, K. Velusamy, M. Gokhale, and R. Pearce, "Metall: A persistent memory allocator for data-centric analytics," *arXiv preprint arXiv:2108.07223*, 2021.

[13] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: astronomical or genomical?" *PLoS Biology*, vol. 13, no. 7, p. e1002195, 2015.

[14] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage." in *USENIX FAST*, vol. 2, 2002, pp. 89–101.

[15] P. Nath, B. Urgaonkar, and A. Sivasubramaniam, "Evaluating the usefulness of content addressable storage for high-performance data intensive applications," in *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, 2008, pp. 35–44.

[16] "Pagemap, from the userspace perspective," https://www.kernel.org/doc/Documentation/vm/pagemap.txt, (Accessed on 03/29/2021).

[17] "Memory-mapped i/o," https://www.gnu.org/software/libc/manual/html_node/Memory_002dmapped-I_002fO.html, (Accessed on 07/06/2021).

[18] K. Youssef, K. Iwabuchi, W. Feng, M. Gokhale, and R. Pearce, "Towards optimizing memory mapping of persistent memory in umap.(abstract)." in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020.

[19] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," *ACM Transactions on Storage (TOS)*, vol. 15, no. 4, pp. 1–40, 2020.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE