Optimizing Performance and Storage of Memory-Mapped Persistent Data Structures

Karim Youssef*†, Abdullah Al Raqibul Islam†‡, Keita Iwabuchi†, Wu-chun Feng*, and Roger Pearce†
*Dept. of Computer Science, Virginia Tech — Email: {karimy,wfeng}@vt.edu
‡Dept. of Computer Science, University of North Carolina at Charlotte — Email: aislam6@uncc.edu
†Center for Applied Scientific Computing, Lawrence Livermore Nat'l Lab — Email: {iwabuchi1,rpearce}@llnl.gov

2022 IEEE High Performance Extreme Computing Conference (HPEC) | 978-1-6654-9786-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/HPEC55821.2022.9926392

Abstract—**Persistent data structures represent a core component of high-performance data analytics. Multiple data processing systems persist data structures using memory-mapped files. Memory-mapped file I/O provides a productive and unified programming interface to different types of storage systems. However, it suffers from multiple limitations, including performance bottlenecks caused by system-wide configurations and a lack of support for efficient incremental versioning. Therefore, many such systems only support versioning via full-copy snapshots, resulting in poor performance and storage capacity bottlenecks. To address these limitations, we present *Privateer 2.0*, a virtual memory and storage interface that optimizes performance and storage capacity for versioned persistent data structures. *Privateer 2.0* improves over the previous version by supporting userspace virtual memory management and block compression. We integrated *Privateer 2.0* into Metall, a C++ persistent data structure allocator, and LMDB, a widely-used key-value store database. *Privateer 2.0* yielded up to 7.5× speedup and up to 300× storage space reduction for Metall incremental snapshots and 1.25× speedup with 11.7× storage space reduction for LMDB incremental snapshots.**

I. Introduction

The ubiquity of large-scale data analytics and the exponential growth in dataset sizes necessitate continuous design and enhancement of scalable data processing tools. Data-analytic workflows include ingesting raw data into different data structures that go through transformations and queries. Persisting these data structures beyond the scope of a single analytic run is a crucial design goal [1]. Data-structure persistence provides two benefits: (1) avoidance of the cost of raw data ingestion for subsequent analytics and (2) consistent data-structure views for incrementally evolving data.

Designing multi-versioned persistent data structures requires optimizing the trade-off between application performance, storage footprint, and programming productivity. To balance this trade-off, multiple data processing systems use memory-mapped I/O (i.e., *mmap*) [1]–[5]. Memory-mapped I/O offers many advantages to persistent data structures when leveraging high-throughput and low-latency storage devices, e.g., non-volatile memory (NVM) [1], [2]. Specifically, memory-mapped I/O incurs no cost for data already in memory and eliminates copies between the operating system's kernel and the application's virtual memory space [6]. Moreover, memory-mapped I/O provides a unified interface to different storage types in complex and multi-tiered storage environments [7].

However, memory-mapped persistent data structures do suffer from several performance limitations, including a (1) lack of scalability with massively multi-threaded applications [6], (2) lack of flexibility for application-specific tuning [7], and (3) write amplification caused by frequent eviction of dirty pages based on system-wide configurations [2]. Moreover, memory-mapped persistent data structures lack efficient versioning support. Existing solutions rely on either full-copy snapshots, which are *not* storage-efficient [1], [4], or log and replay techniques that suffer from high snapshot-reconstruction overhead [5]. Efficient versioning must balance the trade-off between application performance and storage footprint.

To address these challenges, *Privateer* provides an *mmap* alternative that uses private, copy-on-write memory mapping and provides application-controlled writeback [8]. This design enables applications to overcome the kernel's aggressive eviction and control writebacks according to application needs. *Privateer* also incorporates incremental snapshots support via immutable blocks and de-duplication. However, to optimize the storage footprint of snapshots, applications need to use smaller block sizes, which leads to performance degradation.

To address *Privateer*'s limitations, we present *Privateer 2.0*, an enhancement to *Privateer* that incorporates userspace paging and block compression. *Privateer 2.0* provides a more tunable environment to optimize the trade-off between performance and storage footprint. We also integrate *Privateer 2.0* into Metall [1], a C++ persistent data structure allocator, and LMDB [4], a widely-used key-value store database. For LMDB, *Privateer 2.0* enables efficient incremental backups, a desirable feature that is currently unsupported. Our evaluation shows that *Privateer 2.0* achieves up to 7.5× speedup and up to 300× storage reduction for Metall snapshots. *Privateer 2.0* also improves throughput by 1.5× and reduces the storage footprint for storing incremental snapshots of an LMDB database by 11.7×. In summary, our contributions are as follows:

- *Privateer 2.0*, an enhancement to *Privateer* that supports userspace paging and block compression.
- A tunable and productive datastore interface that optimizes the trade-off between application performance and storage footprint of versioned persistent data structures.

- Integration and evaluation of *Privateer 2.0* with state-of-the-art and widely-used data processing systems.

Next, §II explains the design and implementation of *Privateer 2.0*. §III evaluates *Privateer 2.0* with different persistent data-structure allocators. §IV presents related work. §V discusses future directions, and §VI concludes.

II. Design and Implementation of Privateer 2.0

Privateer 2.0 seeks to optimize the trade-off between application performance, storage utilization, and programming productivity for multi-versioned persistent data structures. Existing solutions leverage memory-mapped I/O (i.e., *mmap*) with the MAP_SHARED flag [9] for programming productivity. This approach lets the operating system transparently handle paging and writeback to persistent storage. However, *mmap* is designed and optimized for generality. Specifically, in Linux systems, *mmap* triggers eviction when as low as ten percent of the pages become dirty, which causes write amplification for a wide range of applications [2]. Moreover, writing data to persistent storage at times not defined by the application would affect transaction consistency, especially in case of system failures. Furthermore, *mmap*-based datastores do not allow storage optimization via data compression [9].

The earlier version of *Privateer*, i.e., *Privateer 1.0*, addressed these limitations by leveraging private, copy-on-write, memory mappings (i.e.,MAP_PRIVATE) and explicit buffer flushing calls (i.e., `msync`) that identifies and writes dirtied pages using Linux's */proc/self/pagemap* information. Moreover, *Privateer 1.0* optimized storage utilization by partitioning the mapped datastore into blocks with configurable size and de-duplication. However, *Privateer 1.0* suffered from three limitations. First, de-duplication requires using smaller block sizes to reduce the storage footprint, resulting in performance degradation caused by excessive file-system operations. Second, the `msync` approach incurs kernel page-fault overheads and is only supported on Linux systems. Third, allocating memory using MAP_PRIVATE is bound by physical DRAM size and does not support out-of-core processing.

To address these limitations, *Privateer 2.0* supports userspace paging by capturing the page fault signal using the Linux `sigaction` system call [10]. Userspace paging enables applications to control eviction of dirty pages only via explicit `msync` calls. Userspace paging also enables in-memory block compression, which, when combined with de-duplication, significantly reduces the storage footprint at larger *Privateer* block sizes, thus enabling the simultaneous optimization of application performance and storage footprint. Compression also optimizes I/O bandwidth, especially for network-based file systems, since compressed blocks are loaded into memory and de-compressed in memory. Therefore, *Privateer 2.0* exposes a multi-dimensional and multi-objective tuning space that fits application-specific needs.

A. Privateer 2.0 Architecture

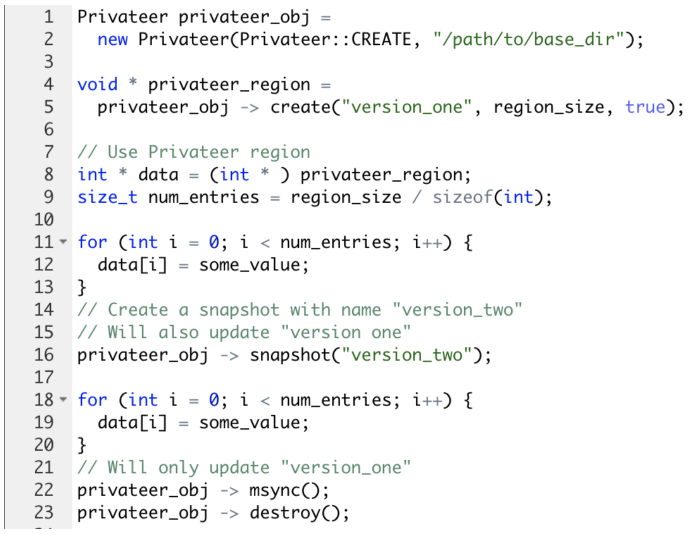
1) APIs: Privateer 2.0 provide APIs for creating, opening, flushing (i.e., `msync`), and snapshotting a virtual memory region. *Privateer 2.0* supports snapshots by means of named versions. A *Privateer* version consists of a metadata recipe file that describes the *Privateer* blocks and their order. Since blocks are immutable, the recipe file contains all the information needed to re-construct a specific snapshot. The snapshot technique is described in further detail later. Figure 1 shows an example usage of *Privateer 2.0*.

2) Userspace Virtual Memory Management: Privateer 2.0 manages virtual memory in userspace to enable handling compressed blocks and to support out-of-core processing while overcoming the limitation of system-managed *mmap*. *Privateer*'s virtual memory manager initially allocates virtual memory space using an anonymous memory mapping with neither read nor write permission (i.e., PROT_NONE). This initialization causes any read or write attempt to trigger a `SIGSEGV` signal that is captured by *Privateer*'s fault handler.

The fault handler identifies the faulting address, the fault type (i.e., read or write), and the *Privateer* block to which the faulting address belongs. The block is then fetched from persistent storage by the block storage manager and permission on that block is updated according to the fault type. *Privateer*'s fault handler adopts a least-recently-used (LRU) eviction policy by maintaining three core lists: the present blocks, clean LRU, and dirty LRU. Eviction is determined by a configurable parameter that determines the maximum in-memory buffer size. If the buffer is full while handling a block fault, the first block of the dirty LRU is evicted by writing its content to a temporary stash directory on persistent storage. When `msync` is called, stashed blocks are un-stashed and moved to the final blocks directory. The block storage manager handles renaming, moving, and writing stashed and present blocks using de-duplication and optionally compression, as elaborated below.

3) Block Storage: Privateer's block storage manager handles block I/O to and from the backing store. As shown in Figure 2, a *Privateer 2.0* datastore consists of a base directory that contains sub-directories for blocks, stash, and versions. The blocks directory contains committed blocks that are named by means of the SHA-256 hash of their content for de-duplication support. The stash directory contains temporary evicted blocks that are named by means of a universally unique identifier (UUID). The `msync` call then renames the evicted and stashed blocks to the SHA-256 hash of their content and moves them to the blocks directory. The block storage manager provides functions used by the virtual memory manager to stash or commit *Privateer* blocks.

The block storage manager also supports block compression. When enabled, *Privateer 2.0* compresses blocks in-memory before writing them to the backing store. It also reads compressed blocks and de-compresses them in-memory. Compression further optimizes storage utilization, especially in the case of incremental snapshots. Without compression, de-duplication requires smaller block sizes to optimize storage. Smaller blocks caused performance degradation in *Privateer 1.0* due to excessive block hash computations and file-system operations to rename blocks. With compression, applications



```
Privateer privateer_obj =
  new Privateer(Privateer::CREATE, "/path/to/base_dir");

void * privateer_region =
  privateer_obj -> create("version_one", region_size, true);

// Use Privateer region
int * data = (int * ) privateer_region;
size_t num_entries = region_size / sizeof(int);

for (int i = 0; i < num_entries; i++) {
  data[i] = some_value;
}
// Create a snapshot with name "version_two"
// Will also update "version one"
privateer_obj -> snapshot("version_two");

for (int i = 0; i < num_entries; i++) {
  data[i] = some_value;
}
// Will only update "version_one"
privateer_obj -> msync();
privateer_obj -> destroy();
```

Fig. 1: Example code demonstrating the use of *Privateer 2.0* to allocate, update, and snapshot virtual memory regions.

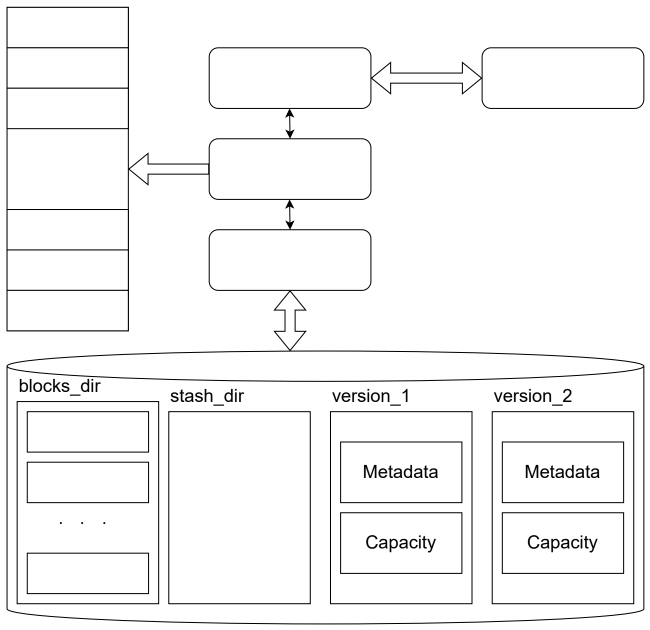


Fig. 2: An overview of *Privateer 2.0* system architecture and directory structure.

can use larger block sizes, which enables storage optimization without a significant loss of performance.

The block storage manager also supports multi-tiered block storage. The stash directory could reside on a different, and ideally faster, storage tier and store committed blocks that are likely to be re-used in the near future. For instance, an application could be writing blocks to a node-local NVMe device while committing blocks to a remote Lustre file system. If an application re-uses blocks in close temporal proximity, the storage manager will look for blocks under the faster tier first. This scenario is useful for HPC setups where node-local storage is not permanent and needs remote store persistence, while multiple job stages are also re-using blocks in close temporal proximity and could benefit from faster store tiers.

B. Optimization Space

The new *Privateer 2.0* design exposes an optimization space with multiple tunable parameters. These parameters include the *Privateer* block size, the use of de-duplication, and the use of compression. Tuning different combinations of these parameters according to an application's need optimizes the trade-off between application performance and storage utilization. Another principal dimension is the data structure design and characteristics. Some persistent data structures would benefit from de-duplication at larger block sizes, while other data structures require compression to optimize storage utilization at larger block sizes. This parameter space enables design optimizations for scalable and efficiently-snapshottable persistent data structures and, in turn, scalable data analytics tools. Our experimental evaluation shows the effect of these parameters for different application domains.

III. Evaluation

As described in §II, *Privateer 2.0* enables the exploration of a multi-dimensional design space. The design objectives are to optimize performance and storage utilization of multi-versioned persistent data structures. Simultaneously optimizing these two objectives is a challenging task [11]. The multi-dimensional design space consists of the following tunable parameters: (a) the Privateer block size, (b) the use of compression, and (c) the data structure characteristics.

Our experiments aim to evaluate the efficiency of *Privateer 2.0* in achieving the aforementioned objectives by tuning the design parameters. Thus, we integrated Privateer into two open-source persistent data-structure tools, Metall and LMDB. We evaluated and compared the performance and storage efficiency of these tools with and without Privateer using three different applications: (1) incremental graph snapshots using the *Wikipedia* page reference graph, (2) incremental snapshots of different Metall C++ data structures using synthetic data, and (3) incremental backups of an LMDB datastore using *Reddit* comments data.

A. Privateer-Enabled Tools

1) Metall: Similar to [8], we integrated *Privateer 2.0* into the latest version of Metall by modifying its data management layer. Specifically, we replaced the *mmap()*, *msync()*, and *snapshot()* APIs with Privateer APIs. Metall users could select whether to use Privateer when compiling their applications.

2) LMDB: LMDB is a key-value datastore that is widely used for processing massive datasets in deep-learning training tasks [12]. LMDB is based on a B+-tree persistent data structure and uses *mmap()* for virtual memory and storage management. We integrated Privateer into the latest version of LMDB and evaluated it using PyLMDB, a Python interface to the LMDB C library.

B. Applications

1) Incremental Graph Snapshots: We used the incremental graph construction and snapshot benchmark reported in [8]. The benchmark consists of ingesting a raw edge list that represents a time-evolving graph into a Metall persistent C++ adjacency list and storing temporal snapshots. In our experiment, we ingested the *Wikipedia* page reference graph and stored snapshots after inserting each year's data (a total of 17 snapshots). The dataset is described later in this section.

2) LMDB Database Incremental Construction and Backup: To evaluate LMDB with *Privateer 2.0*, we implemented a benchmark that incrementally constructs an LMDB key-value datastore of *Reddit* comments. Each key consists of a comment ID, and its value consists of the comment body. The benchmark stores a snapshot of the database after inserting each month's comments from the 2017 data. The current version of LMDB only supports full-snapshot backups. We evaluated *Privateer*'s improvement over LMDB's backup functionality in terms of throughput and total size of snapshots.

3) Incremental C++ STL Snapshots: We used a similar setup to the incremental graph construction to evaluate several C++ standard template library (STL) containers (e.g., vector, list, deque, map, set, etc.) with Metall and *Privateer 2.0*. We implemented a benchmark that stores incremental snapshots of each data structure. In our evaluation, we used two workloads: (i) *sequential* and (ii) *random*. For both of the workloads, we inserted 100 million entries (64-bit signed integers) in total and incrementally created a snapshot after one-million insertions (100 snapshot in total). In the *sequential* workload, we sequentially inserted numbers (from 0 to 100 million) in the containers. In the *random* workload, we generated random numbers within the 64-bit signed integer range using a *uniform distribution* (`std::uniform_int_distribution`).

C. Experiment Setup

1) Computing Platform: We conducted our evaluations on Mammoth, a high-performance computing cluster that consists of 64 nodes. Each node is equipped with two AMD EPYC 7742 64-core processors, 2TB of memory, and 3TB of NVMe SSDs.

2) Datasets: We evaluated *Privateer 2.0* using two real-world datasets previously used in [1]: (1) The *Wikipedia* page reference graph, which consists of 1.8 Billion edges with a raw size of 13GB, and (2) the *Reddit* comments dataset, which consists of 4.4 Billion comments with a raw size of 1.9TB.

D. Results

1) Incremental Graph Snapshots: We evaluated the throughput and total size of snapshots for incrementally constructing and snapshotting a Metall adjacency list data structure built from the *Wikipedia* page reference graph. Our benchmark stores snapshots of the evolving graph after inserting each year's data (a total of 17 snapshots). We compared the baseline (i.e., *Metall* without *Privateer*) to *Privateer 1.0* and *Privateer 2.0* with different *Privateer* block sizes. We compared two different configurations of *Privateer 2.0* — with and without compression. Moreover, we compared two adjacency list implementations, a hash-table-based unordered map, and a self-balancing binary search tree (BST)-based ordered map. Figure 3 shows the throughput results while Figure 4 shows the total size of the snapshots.

For the hash-table based unordered map, *Privateer 1.0* and *Privateer 2.0* yielded performance gains at larger Privateer block sizes. For the 32MB-block size, *Privateer 2.0 without* compression yielded a 1.58× speedup over the baseline Metall, while *Privateer 2.0 with* compression yielded a 1.14× speedup over the baseline Metall. Storage optimization occurred at smaller block sizes without compression. This observation is due to the unordered map's rehash operation, which causes a significant restructuring, leading to dirtying a larger number of bytes unnecessarily. Hence, smaller block sizes were required to benefit from de-duplication. *Privateer 2.0* with compression yielded significant storage optimization at both smaller and larger block sizes, hence optimizing the trade-off between performance and storage efficiency.

Conversely, for the BST-based ordered map, *Privateer* yielded both performance gains and storage optimizations at smaller block sizes. Compression further optimized storage at the cost of performance loss.

2) LMDB Incremental Snapshots: Figure 5 shows the evaluation of LMDB's incremental backup using monthly backups of the 2017 *Reddit* comments dataset. While LMDB provides an efficient and relatively fast *copy* interface, LMDB+Privateer yielded nearly 1.5× higher throughput because it overcomes slowdowns caused by *mmap*. This improvement amortized *Privateer*'s compression overhead, yielding both throughput and storage footprint improvement. Using a *Privateer* block size of 32MB and block compression, LMDB+Privateer reduced the storage footprint by 11.7× when compared to baseline LMDB. It was also observed that using *Privateer* without compression yielded comparable throughput to using compression. Because LMDB uses a single writer and because old blocks do not get updated very often in this application, *Privateer*'s multi-threaded `msync` optimizes compression overhead with minimal `sigaction` thread serialization overhead.

3) Incremental STL Snapshots: We evaluated the runtime and the storage footprint of incrementally constructing and snapshotting seven C++ STL data structures on two workloads, as mentioned earlier. We compared the baseline (i.e., *Metall* without *Privateer*) to *Privateer 2.0* with and without compression across different *Privateer* block sizes. Table I lists the runtime and the storage footprint of STL data structures in our baseline model (i.e., *Metall* without *Privateer*), while Figure 6 shows the comparative storage and runtime improvement in the different setup of *Privateer 2.0* compared to the baseline model. We used the local NVMe SSDs to save all the data and snapshots.

With the *random* workload, *Privateer 2.0* with compression achieved up to 53.57× better storage space utilization, while *Privateer 2.0* without compression achieved 29.04× better storage space utilization, relative to baseline Metall, as shown in Figure 6(a). Furthermore, no data structure acquired more

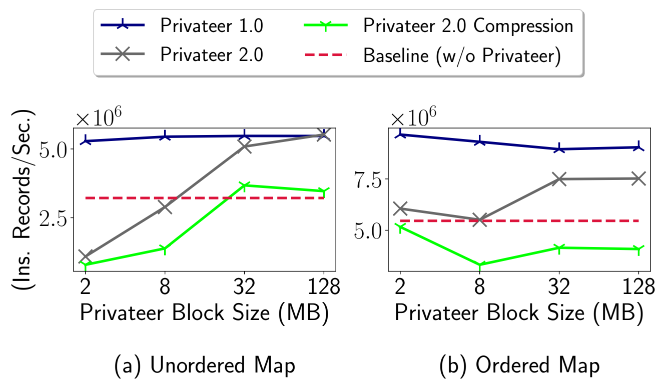


Fig. 3: Throughput (higher is better) of incrementally ingesting the *Wikipedia* page reference graph into a Metall adjacency list and storing yearly snapshots. The adjacency list is implemented using (a) an unordered hash-based map of vectors, and (b) an ordered BST-based map of vectors.

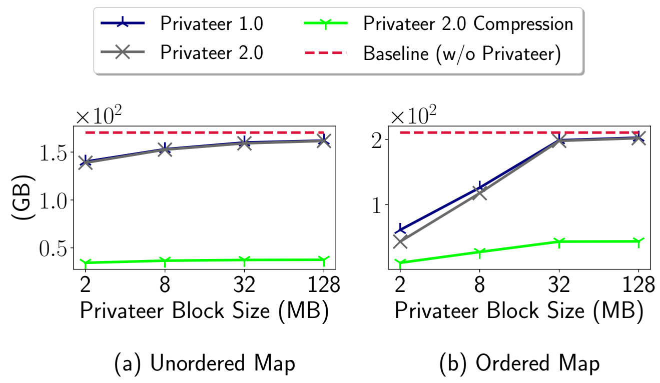


Fig. 4: Total size (lower is better) of incrementally ingesting the *Wikipedia* page reference graph into a Metall adjacency list and storing yearly snapshots. The adjacency list is implemented using (a) an unordered hash-based map of vectors, and (b) an ordered BST-based map of vectors.

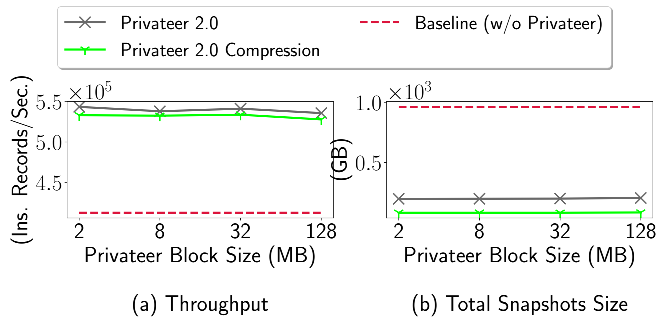


Fig. 5: Evaluation of (a) throughput (higher is better) and (b) total datastore size (lower is better) for incremental construction and storage of monthly snapshots of an LMDB database from *Reddit* comments of the year 2017.

space while increasing the *Privateer* block size (up to 128 MB). These results demonstrate the strength of *Privateer 2.0*'s block management. So, *Privateer 2.0* outperforms the baseline *Metall* with respect to both storage footprint and runtime. For runtime, *Privateer 2.0* with compression achieves up to 13.71× speedup over baseline Metall, and *Privateer 2.0* without compression achieves up to 14.45× speedup over baseline Metall, as shown in Figure 6(c). In addition, *Privateer*'s de-duplication helps reuse the blocks and mutually benefits space and runtime. Also, compression leads to better storage utilization with only a minor increase in runtime overhead.

TABLE I: Performance and snapshot sizes of STL containers using *Metall* without *Privateer 2.0* (baseline).

	Random Workload		Sequential Workload	
STL Containers	Snapshot Size (GB)	Runtime (Sec.)	Snapshot Size (GB)	Runtime (Sec.)
Vector	43	43.56	43	24.99
List	122	72.25	122	51.25
Deque	45	37.09	45	22.45
Map	198	670.78	198	158.00
Unordered Map	217	505.40	217	198.31
Set	160	568.25	160	168.12
Unordered Set	181	512.89	181	193.26

Similarly, in sequential workload, *Privateer 2.0* achieved better performance in both storage space (up to 300× with compression and 37.5× w/o compression; showing in figure 6(b)) and runtime (up to 7.54× with compression and 10.13× w/o compression; as shown in figure 6(d)). One interesting observation is that *Privateer 2.0* shows huge storage space improvement in sequential workload compared to the random (e.g., 300× in sequential Vs. 53.57× in random workload). This improvement is because we inserted data sequentially from 0 to 100 million in sequential workload, but data is saved as a 64-bit integer. The compression in *Privateer 2.0* handles such repeated patterns in the data very efficiently. Another important observation is that, for all the cases in figure 6, *Privateer 2.0*'s performance gradually decreases while increasing the *Privateer* block size. The reason is that a larger block size leads to dirtying a larger number of bytes unnecessarily between snapshot creation; hence, a smaller block size performs better.

IV. Related Work

The benefits and drawbacks of memory-mapped I/O for persistent data structures have been studied extensively. We discuss related work in terms of state-of-the-art memory-mapped persistent data structures, existing solutions to optimize memory-mapped I/O, and the incremental backup problem for different types of persistent data stores.

Data-structure persistence is a core component of high-performance data analytics. Memory-mapped I/O has been widely used for different types of persistent data structures, such as key-value datastores, graph processing engines, and generic C++ data strucutres. Memory-mapped key-value datastores include LMDB [4], RocksDB [11], Kreon [2], CedrusDB [5] and MongoDB [3]. Key-value datastores are built on top of different types of persistent data structures, including

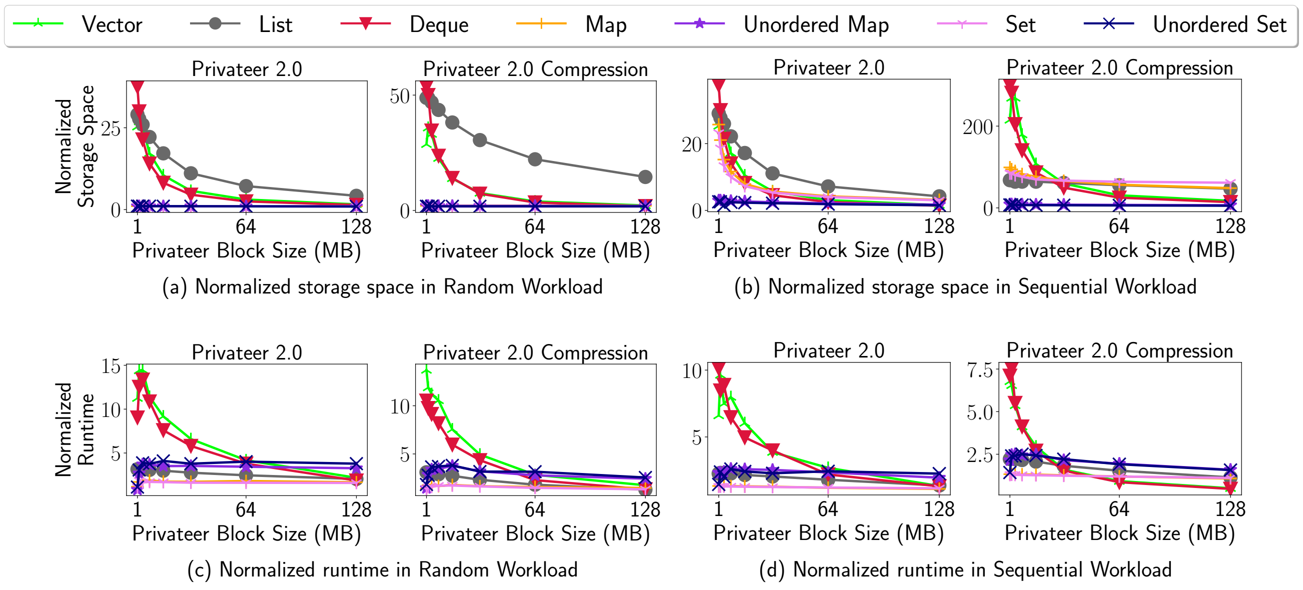


Fig. 6: Improvement of STL container snapshots (higher is better) by *Privateer 2.0* (w/ and w/o compression) compared to the baseline Metall. Fig. (a) and (b) show relative storage space improvement in random and sequential workloads respectively. Fig. (c) and (d) show relative runtime improvement in random and sequential workloads respectively.

B+trees [13], LSM-trees [14], and lazy tries [5]. Graph engines that leverage memory-mapped I/O include LLAMA [15] and HavoqGT [16]. Recently, Metall emerged as a generic C++ persistent data structure allocator [1]. Metall enables persisting standard C++ data structures beyond the scope of a single execution. Metall has been used in multiple domains, including graph processing [16] and scalable distributed data processing [17]. Metall uses *mmap* to manage virtual memory and backing store for persisting data structures.

These tools have either used the system's *mmap* without changes or optimized versions of it using different techniques. These techniques include optimized kernel-level *mmap* path [2], [6], userspace memory-mapped I/O [7], [18], [19], or a gradual step-away from using *mmap* into implementing userspace buffers to avoid mmap's bottlenecks [11], [20]. Kernel-space mmap optimizations require changing or updating operating system modules. Moreover, they limit applications to global system-wide configurations. These limitations motivated the design of userspace memory-mapped I/O. *Privateer 2.0* improves over existing userspace solutions by supporting incremental versioning, compression, and future support for multi-tiered block storage.

Storage optimization using de-duplication and/or compression has been explored in file systems [21], cloud backup [22], and incremental backups [23]. *Privateer 2.0* combines these techniques with userspace virtual memory management to provide a snapshot-friendly data management system that enables a novel optimization space for data-structure persistence in high-performance data analytics.

V. Future Directions

Future directions include deeper explorations of the *Privateer* optimization space for different application domains. For instance, integrating and evaluating *Privateer* with different key-value datastore designs and performing application or domain-specific optimization of the trade-off between throughput and storage efficiency. Other future directions include more efficient support of multi-tiered block storage via multi-tiered paging and intelligent multi-tiered block movement. Also, improving *Privateer*'s multi-threading support by leveraging Linux's *Userfaultfd* [24] to overcome `sigaction`'s serialization of threads, and hence improve scalability for massively multi-threaded applications. Finally, another future direction is to support distributed and disaggregated block storage.

VI. Conclusion

Privateer is virtual memory and storage management tool that optimizes the trade-off between application performance, storage footprint, and programming productivity for persistent data structures. In this paper, we presented *Privateer 2.0*, an upgrade to *Privateer* that exposes a more tunable environment by incorporating userspace page fault handling and block compression. *Privateer 2.0* enables optimizing the storage footprint of incremental snapshots without significant performance loss. We integrated *Privateer 2.0* into Metall, a C++ persistent data structure allocator, and LMDB, a widely-used persistent key-value datastore. *Privateer 2.0* achieved 7.5× and 1.5× performance improvement for Metall+Privateer, and LMDB+Privateer, while achieving up to 300×, and 11.7× storage footprint reduction for Metall and LMDB, respectively.

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-839424). 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.

References

[1] K. Iwabuchi, K. Youssef, K. Velusamy, M. Gokhale, and R. Pearce, "Metall: A persistent memory allocator for data-centric analytics," *Parallel Computing*, vol. 111, p. 102905, 2022.

[2] 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.

[3] D. Hows, P. Membrey, E. Plugge, and T. Hawkins, "Introduction to mongodb," in *The Definitive Guide to MongoDB*. Springer, 2015, pp. 1–16.

[4] G. Henry, "Howard chu on lightning memory-mapped database," *Ieee Software*, vol. 36, no. 06, pp. 83–87, 2019.

[5] M. Yin, H. Zhang, R. van Renesse, and E. G. Sirer, "Cedrusdb: Persistent key-value store with memory-mapped lazy-trie," *arXiv preprint arXiv:2005.13762*, 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. B. Peng, M. B. Gokhale, K. Youssef, K. Iwabuchi, and R. Pearce, "Enabling scalable and extensible memory-mapped datastores in userspace," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 866–877, 2021.

[8] K. Youssef, K. Iwabuchi, W.-C. Feng, and R. Pearce, "Privateer: Multi-versioned memory-mapped data stores for high-performance data science," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–7.

[9] A. Crotty, V. Leis, and A. Pavlo, "Are you sure you want to use mmap in your database management system?" in *CIDR 2022, Conference on Innovative Data Systems Research*, vol. 64, 2022, pp. 2195–2207.

[10] D. B. Thangaraju, "Linux signals for the application programmer," *Linux Journal*, vol. 2003, no. 107, p. 6, 2003.

[11] 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.

[12] S. Pumma, M. Si, W.-C. Feng, and P. Balaji, "Scalable deep learning via i/o analysis and optimization," *ACM Transactions on Parallel Computing (TOPC)*, vol. 6, no. 2, pp. 1–34, 2019.

[13] G. Graefe, "Modern b-tree techniques," *Foundations and Trends® in Databases*, vol. 3, no. 4, pp. 203–402, 2011.

[14] 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.

[15] 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.

[16] R. Pearce, "Highly asynchronous visitor queue graph toolkit," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2012.

[17] P. Pirkelbauer, S. Bromberger, K. Iwabuchi, and R. Pearce, "Towards scalable data processing in python with clippy," in *2021 IEEE/ACM 11th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 2021, pp. 43–52.

[18] 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*, vol. 18, no. 1, pp. 15–28, 2015.

[19] S. Rivas-Gomez, A. Fanfarillo, S. Valat, C. Laferriere, P. Couvee, S. Narasimhamurthy, and S. Markidis, "uMMAP-IO: User-level memory-mapped I/O for hpc," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019, pp. 363–372.

[20] M. D. Da Silva and H. L. Tavares, *Redis Essentials*. Packt Publishing Ltd, 2015.

[21] S. Quinlan and S. Dorward, "Venti: A new approach to archival data storage," in *Conference on File and Storage Technologies (FAST 02)*, 2002.

[22] D. N. Simha, M. Lu, and T.-c. Chiueh, "A scalable deduplication and garbage collection engine for incremental backup," in *Proceedings of the 6th International Systems and Storage Conference*, 2013, pp. 1–12.

[23] Z. Zhang, H. Hu, Z. Xue, C. Chen, Y. Yu, C. Fu, X. Zhou, and F. Li, "Slimstore: A cloud-based deduplication system for multi-version backups," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1841–1846.

[24] "userfaultfd(2) — linux manual page," https://man7.org/linux/man-pages/man2/userfaultfd.2.html, accessed: 2022-08-23.