2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10071009

Multi-Granularity Shadow Paging with NVM Write Optimization for Crash-Consistent Memory-Mapped I/O

Hongchao Du*, Qiao Li†, Riwei Pan*, Tei-Wei Kuo‡, Chun Jason Xue*
*Department of Computer Science, City University of Hong Kong
†Department of Computer Science and Technology, Xiamen University
‡Department of Computer Science and Information Engineering, National Taiwan University

***Abstract*—The complex software stack has become the performance bottleneck of the system with high-speed Non-Volatile Memory (NVM). Memory-mapped I/O (MMIO) could avoid the long-stack overhead by bypassing the kernel, but the performance is limited by existing crash-resilient mechanisms. We propose a Multi-Granularity Shadow Paging (MGSP) strategy, which smartly utilizes the redo and undo logs as shadow logs to provide a light-weight crash-resilient mechanism for MMIO. In addition, a multi-granularity strategy is designed to provide high-performance updating and locking for reducing runtime overhead, where strong consistency is preserved with a lock-free metadata log. Experimental results show that the proposed MGSP achieves $1.1 \sim 4.21\times$ performance improvement with write and $2.56 \sim 3.76\times$ improvement with multi-threads write compared with the underlying file system. For SQLite, MGSP can improve the database performance by 29.4% for Mobibench and 36.5% for TPCC, on average.**

I. Introduction

The advances in emerging Non-Volatile Memories (NVM) like PCM [11], [24], [34], STT-RAM [22] and 3D Xpoint memory [1], could significantly reduce I/O access latency to sub-microsecond levels, outperforming traditional block-based devices [20]. There are two mainstream methods to access data on NVM. The first is to access NVM through the traditional software I/O stack, which can provide the required data consistency by the file system. However, the close-to-DRAM access latency of NVM devices makes the software overhead in I/O stack no longer negligible [5], [7], [14], [38], [40], [43]. Fortunately, the byte-addressability of NVM devices presents the potential for applications to directly access (DAX) the file data on NVM, bypassing the file system I/O stack [27], [39]. Memory-mapped I/O (MMIO) takes full advantage of the raw performance of NVM devices by mapping the file data into the process' address space to access persistent data like accessing memory. The price of high performance is the need for additional crash consistency strategies to ensure data correctness. Inappropriate designs could result in performance degradation or data corruption [21], [41]. Therefore, a proper crash-consistent mechanism is critical for the MMIO on NVM.

There are two main strategies for MMIO to provide data consistency: shadow paging [12], [18], [30], [41], [42] and logging [13], [14], [17], [25], [29], [32], [33]. Shadow paging writes the data on new blocks and then redirects the pointer of the original data to the new blocks. Shadow paging could suffer from severe write amplification problems when the access granularity is small [10]. In addition, shadow paging needs to constantly modify the data pointer, which could cause TLB-shootdown and affect performance when using MMIO [3], [37]. Logging writes a data log before updating the file data to ensure that the system can recover after a crash. The main drawback of logging is the double-write problem, one write for the log and one write for the actual data [9], [14], [31]. Meanwhile, logging with fixed units like a page or a block also has the write amplification problem, where the entire data block must be logged even if only a small part is updated. Libnvmmio [9] designs the failure-atomic MMIO based on user-level logging. To achieve the best performance of redo and undo logging, Libnvmmio provides hybrid logging and uses background threads to move double writes away from the critical path. Even though write amplification is alleviated with differential logging, the double-write problem still exists with frequent sync operations.

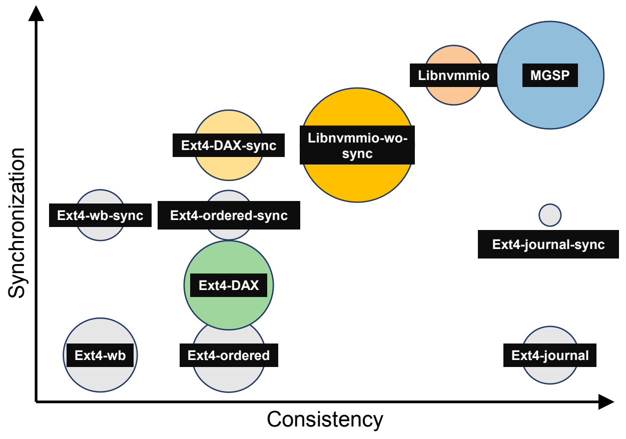

Fig. 1. 4KB write performance of file systems under different consistency and synchronization requirements. Larger sizes represent higher performance.

Figure 1 shows the performance of Ext4 file system with different consistency strategies. The Ext4 file system supports three consistency modes: `writeback` (`wb`), `ordered`, and `journal`, with progressively stronger consistency protection but no guarantee that updates are synchronized. We execute `fsync` after each operation in Ext4 and label them with `-sync`. Manually performing sync can improve synchronization, but at the expense of performance. Ext4-DAX can write data directly to storage devices but does not support `journal` mode. Although Libnvmmio can ensure better data consistency and performance when no sync operation is performed, the performance will drop significantly after adding sync. To achieve both benefits, this paper presents a high-performance crash-consistent mechanism for NVM-based MMIO with strong consistency and synchronization.

We propose Multi-Granularity Shadow Paging (MGSP), a novel crash consistency mechanism for DAX-MMIO. MGSP introduces the concept of shadow log, combining the advantages of shadow paging and logging. The shadow log avoids the double-write problem by switching the roles of data blocks between redo and undo logs when needed. To mitigate write amplification problems in fine-grained updates and the metadata overhead in coarse-grained updates, MGSP adaptively chooses the proper granularity of shadow logs for specific accesses. The shadow logs of various granularities are managed with different levels of a radix tree, where on-demand nodes will be created on the radix tree to support different granularities. MGSP uses bitmaps to identify the shadow log's existence and validity and achieve operation atomicity, where a compact lock-free metadata log is designed. In addition to that, MGSP explores scalability by choosing the appropriate lock granularity for accesses of different sizes. Finally, a series of optimization mechanisms are proposed to reduce the search and update overhead of the tree structure without affecting its correctness. Experimental results show that MGSP achieves $1.1 \sim 4.21\times$ write performance improvement on FIO and 36.5% improvement of TPCC on SQLite. MGSP is available at `https://github.com/MIoTLab/MGSP`.

This paper makes the following contributions:

- MGSP proposes shadow logging for crash-consistent MMIO, avoiding double writes in traditional logging.
- MGSP designs the multi-granularity shadow logging to avoid write amplifications from write requests with different sizes.
- MGSP achieves high scalability with lock-free metadata logging and multiple granularity locking.
- Several optimization strategies are further proposed to reduce the management overhead of MGSP.

II. Background and Motivation

A. Direct Access with NVM

Emerging Non-Volatile Memory (NVM) like PCM [11], [24], [34], STT-RAM [22] and 3D Xpoint memory [1], is a new memory technology supporting memory-like access while ensuring durability. The rapid increase of hardware performance introduces new challenges to traditional software design of I/O stack, especially to the design of file systems [14], [41]. The file system architecture in the kernel largely explores the characteristics of block devices, and I/Os must go through the page cache layer, block device layer, and driver layer to reach the storage device, resulting in heavy software overhead to the performance of NVM file systems [21]. Some Direct Access (DAX) file systems bypass the page cache layer to access data directly [27] but cannot ensure strong crash consistency. Therefore, many NVM-aware file systems have been proposed in the past decade [8], [9], [12], [14], [21], [23], [27], [39], [41], [42], to alleviate performance degradation and guarantee crash consistency under the traditional file system architecture. However, the high performance of NVM makes the overhead of context switches and system calls between user space and kernel space non-negligible [9], [21]. An intuitive solution is to move the file system to user space, but this introduces new problems such as stray write, permission management, and atomicity of operations [8], [21]. To address these problems, a series of user-space NVM file systems are introduced [9], [21], [23]. These user-space file systems rely on memory-mapped I/O (MMIO) to directly access NVM devices in user space.

MMIO maps the file data into the application's process address space [10], [26], [35], [36]. Once the mapping is established, data access can be performed through load/store instructions without trapping into kernel space [40]. Though MMIO can achieve direct access on NVM devices and take advantage of the high raw performance, it lacks data consistency protection [29], [44]. Applications must be aware of device-level details and design their crash consistency mechanisms to ensure data correctness [9], [21]. For example, to deal with the cache write reordering problem, applications must decide when to flush the cache and wait for the flush to complete. Improper scheduling of these operations can result in performance degradation or data corruption [9].

B. Crash Consistency Mechanisms

Ensuring data consistency is a fundamental requirement for storage devices. However, most 64-bit CPUs only guarantee atomicity for 8-byte operation, which introduces new challenges in the design of NVM-based file systems [4], [45]. The NVM-based file systems need to flush the cache lines with non-temporal store commands to ensure data persistence [19], [46]. In addition to durability, atomicity and isolation are also required to ensure data consistency [31]. Traditional file systems typically use failure-atomic techniques for atomicity and locking for isolation.

1) Failure-atomic update:

Logging. The logging (or journaling) technique persists a data log before the actual update to ensure that the system can recover after a crash [13], [14], [17], [25], [29], [32], [33]. There are two types of logging: undo and redo, which store the original and new data in the log, respectively. Undo and redo logging has the double-write problem, one write for the log and one write for the actual data [9], [14], [31]. Meanwhile, logging with fixed units like a page or block may cause write amplification. The entire data block must be logged, even if only a small part is updated. Differential logging schemes [2], [15] are proposed only to log the updated data to reduce the log overhead. Libnvmmio [9] proposes hybrid logging, which dynamically adjusts the logging strategy to reduce the log overhead in write-dominant or read-dominant scenarios. In addition, Libnvmmio uses background checkpointing to remove the double write from the critical I/O path as much as possible. Libnvmmio can achieve high crash-consistent MMIO performance, but it adopts fsync-based checkpointing, which degrades Libnvmmio's performance due to the double write and the lock conflict between the foreground and background threads under the scenario in need of frequent sync operations like database.

Shadow paging. Shadow paging or copy-on-write (CoW) [12], [18], [30], [41], [42] technique writes the data on new blocks and then redirects the pointer of original data to the new blocks. Shadow paging can avoid double-write, but write amplification still exists, especially when using large-grained units like the huge page [10]. In addition, shadow paging needs to constantly modify the data pointers, which could cause TLB shootdown and affect the performance when using MMIO [3], [37]. The CoW mechanism on the tree-based indexing structure introduces the wandering tree problem [6], which propagates modifications of data and metadata from leaf nodes to the root node. BPFS [12] proposes Short-Circuit Shadow Paging for CoW on NVM, which uses 8-byte atomic operations to reduce propagating costs as much as possible. NOVA [41] uses CoW to achieve atomic MMAP, but it also results in significant overhead because of the write amplification and TLB shootdown. SplitFS [21] proposes an efficient way to utilize DAX-MMIO in user space and provides flexible crash consistency guarantees. All data operations directly access memory-mapped files with a user-space library. However, the relink operation of SplitFS still requires CoW to ensure strong data consistency, leading to write amplification for small writes.

2) Isolation with locking: In addition to atomicity, isolation is also necessary for data consistency to avoid parallel updates to the same data block [31]. A common practice in file systems is to lock the file before update operations. However, file-level locks do not allow multiple threads to modify a file simultaneously, even if they update different parts of the file. Besides, logging file systems, like Ext4, also require locking the metadata, such as the shared log area, to ensure atomicity [8], limiting the concurrent access performance. Therefore, range locking and fine-grained locking are proposed to improve the scalability [8], [9].

C. Drawbacks of Existing Crash-Consistent MMIO

Since user-space access has natural advantages over kernel-space access, MMIO with a suitable crash-consistent mechanism is a potential way to efficiently access NVM [9], [21], [41]. However, existing NVM-aware file systems cannot fully exploit these advantages. Direct use of logging or shadow paging incurs additional overhead, such as extra writes. Libnvmmio [9] designs the memory mapping interface based on hybrid logging and provides failure-atomicity by extending the semantics of *msync* in user space. However, it cannot provide operational-level consistency and synchronization guarantee as other NVM file systems [21], [41]. The newly written data will be synchronized to the original file only after the sync operation is performed, and the atomicity of a single write operation is not supported. Although Libnvmmio can enhance file consistency by triggering sync more frequently, it can seriously impact system performance.

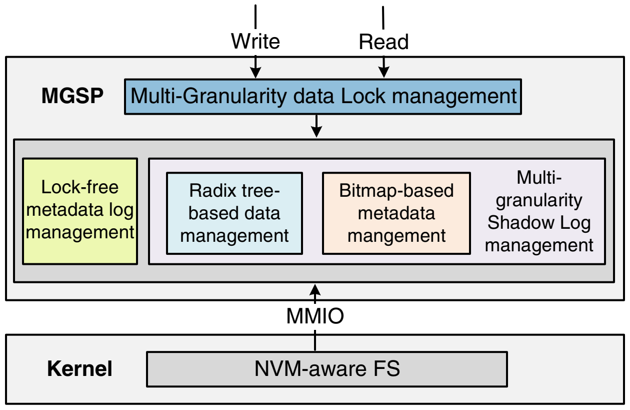
Write
Read
MGSP
Multi-Granularity data Lock management
Lock-free metadata log management
Radix tree-based data management
Bitmap-based metadata mangement
Multi-granularity Shadow Log management
MMIO
Kernel
NVM-aware FS

Fig. 2. MGSP Overview.

III. Multi-Granularity Shadow Paging

A. Overview

This section presents Multi-Granularity Shadow Paging (MGSP), a new crash-consistent mechanism designed for NVM-based MMIO. We propose to utilize shadow logging, which combines the benefits of shadow paging and logging, to implement efficient crash-consistent MMIO. MGSP stores file data in the shadow logs with different granularities, ranging from a data block to an entire file. The multi-granularity design is used to resolve the issues in fixed-granularity management mechanisms, such as write amplification for fine-grained updates and heavy metadata overhead for coarse-grained updates. The multi-granularity design also enables a compact metadata log and an efficient locking mechanism.

The overall architecture of MGSP is shown in Figure 2. MGSP runs as a user-space library and manages the shadow logs via the MMAP of the underlying file system. All data is accessed directly from the shadow log on the NVM devices. When the read and write requests come, MGSP first locks the accessed area through a multi-granularity locking mechanism (§III-C2). By using appropriate locks for operations of different sizes, MGSP achieves high-concurrency and low-overhead locking. Then a radix tree-based method is used to find the shadow logs for read and write with corresponding granularities and writes new logs when needed (§III-B1). The new logs are integrated into the file by modifying the bitmap-based metadata (§III-B2). To ensure the atomicity of the operation, MGSP first writes the metadata log before actually changing the bitmap (§III-C1). Finally, MGSP releases the lock to complete the corresponding operation.

In summary, MGSP includes Multi-granularity Shadow Log (MSL), Multi-Granularity Locking (MGL), and the lock-free metadata log, which will be introduced in the following.

B. Multi-granularity Shadow Log

1) Radix-tree-based multi-granularity data management:

Shadow Logging: The key insight of shadow logging is derived from the following observation on logging strategies. Figure 3 shows the undo/redo log and our proposed shadow log strategy. The core of the logging idea is to keep a copy

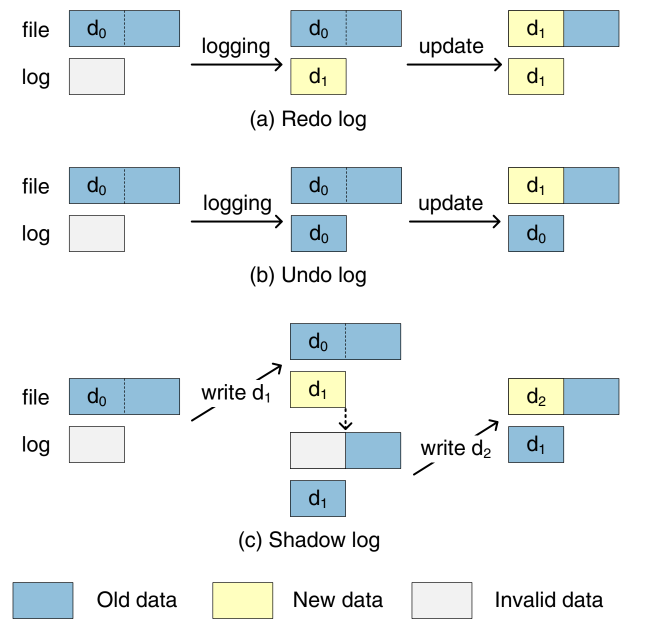

Fig. 3. The comparison between redo log, undo log, and shadow log. Redo and undo logs write two data blocks for one write operation, while shadow logging can complete two write operations by writing two data blocks.

of old or new data in the log before writing file data. For each write request, the redo/undo log will write a new/old data block as the log (the first step) before writing data to files (the second step), which requires writing two data blocks for each write operation. However, for multiple update operations on the same data block, the conversion of old and new data brings the possibility of eliminating duplicated data copies.

Suppose that two write operations are performed to the same data block continuously. First, we adopt redo logging to write the new data d_1 in the log. If the second write operation arrives before the execution of the second step in redo logging, d_1 becomes the old data. At this time, there is a copy of d_1 in the log, which satisfies the conditions for executing the second step of the undo logging, so we can directly write the new data d_2 to the original file block. In the process, we only write two data blocks to complete two write operations and achieve zero copy. We call this kind of log that mixes redo and undo logging as the shadow log.

Shadow log can achieve zero-copy crash consistency under fixed granularity updates, as shown in Figure 3. However, when the update granularity is smaller than the data block size, such as less than 4KB, unmodified data in the updated data block still needs to be copied, resulting in write amplification. When the update granularity is larger than the data block size, consecutive allocations of multiple data blocks will cause significant metadata overhead. To this end, we propose to extend the shadow log to support fine-grained and coarse-grained logging, called Multi-granularity Shadow Log (MSL), to reduce runtime overhead and write amplification.

Multi-granularity Shadow Logging: MSL is responsible for managing shadow logs of different granularities. For each write, MSL will first choose the appropriate granularity for the log according to the write offset and size, then write the data to

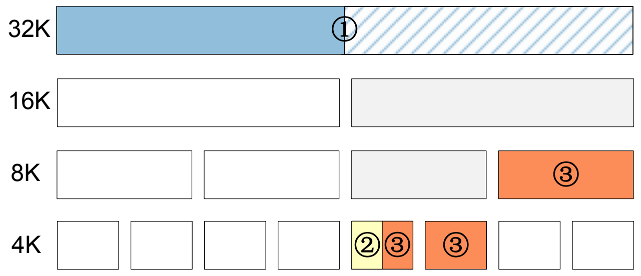

Fig. 4. Multi-granularity Shadow Log (MSL). Colored rectangles represent nodes created on the radix tree. This example shows the data distribution of MSL after three writes to a blank file.

the corresponding shadow log. To fully exploit the advantage of multi-granularity, MSL can use multiple logs with different granularities for one write. A radix tree-based structure is introduced to manage the logs of different granularities. Each level of the tree is responsible for one granularity. From the root node to the leaf node, the granularity of the log decreases sequentially. The root node's log is the file's memory map. Besides the root node, the tree nodes are created only when needed, and all the child nodes together maintain the same data range as their parent node. The maximum level of the radix tree depends on the file size.

When a write operation comes, MSL first checks whether it is beyond the maximum file size supported by the current radix tree. If yes, the level of the radix tree will be extended to support a larger file. Then MSL will start traversing the radix tree from the root node. If the write size is smaller than the current node's granularity, MSL will traverse its child nodes. If one child node is insufficient to cover the operation, the write may be processed by multiple child nodes, each of which handles a part of the write. Traversal stops when it encounters the smallest granularity or when the log granularity of the node is equal to the write size. Algorithm 1 shows how to find the logs and apply the shadow logging with a recursive version of MSL.

Figure 4 shows an example of MSL. Each node has two child nodes, meaning that each child node's granularity is half of the parent node. In this example, we first write ① 32KB to an empty file. The smallest data block is 4KB, so the maximum height of the radix tree for a $32 = 4 \times 2^3$ KB file is 3. The top rectangle represents the shadow log of the root node, which is the mmap of the file itself. Any write to the root node is directly written to the file. The second write ② is to write 2KB data from the 16KB offset. MSL will traverse the radix tree to find the appropriate node for the write, a leaf node in this case. MSL only updates the first 2KB of the 4KB log with fine-grained logging. Fine-grained updates allow MSL to write data smaller than the minimum data block size. In this example, the minimum granularity is 2KB instead of 4KB. There will still be some redundant writes if the write is not 2KB aligned. The smaller the minimum update granularity, the fewer redundant writes, but the management overhead will also increase. We will discuss deciding the minimum update granularity in §III-B2.

MSL uses coarse-grained logs to reduce metadata overhead and improve write performance whenever possible. The third write ③ updates the last 14KB of this file. The coarse-grained logging adaptively selects an appropriate combination of logs with different granularities, two 4K logs and one 8K log for this write. The 4KB log in the second fine-grained write can be reused, so there is no space wasted in this case. The logs do take up extra space outside the file. However, the additional space required for each granularity of logs does not exceed the file size. And this space can be reclaimed when the file is closed.

Both MGSP and Libnvmmio reduce software stack overhead through user space access and use the radix tree to manage logs. The most significant difference between MGSP and Libnvmmio is the idea of shadow logging. By storing data blocks of different granularities in different levels of the radix tree and using them as logs for each other, MGSP avoids double-write and foreground or background checkpointing. In contrast, Libnvmmio only stores logs at the leaf nodes of the radix tree and can only use logs of one granularity (determined at the first write) for the specified data. At the same time, Libnvmmio must write the log back to the file after sync, which has a non-negligible impact on performance, whether in the foreground or the background. In contrast, MGSP can always read the latest data from the log, irrespective of sync. We will discuss it in the next section.

Minimum search tree. As the file size grows, the radix tree would have more levels, which results in higher search overhead. For this reason, MSL uses the minimum search tree to store the subtree that is likely to cover the subsequent requests. Based on the principle of locality, the minimum search tree is set to the smallest subtree that can cover the previous operation. It will be updated if the minimum search tree changes after every operation. Before searching from the root node, MSL first checks whether the current operation can be completed in the cached subtree. If not, MSL will try the next adjacent subtree. If the second attempt fails, MSL will search from the root node.

2) Bitmap-based metadata management: Multi-granularity shadow logs effectively reduce redundant data copies and improve write performance, but the latest data may exist in logs at any level of the radix tree. How to efficiently find the latest data is critical. MSL searches the radix tree for every read and write. Additional metadata is needed to indicate whether a node has the latest data block and whether to stop the search. Efficient metadata querying and updating are the keys to fully exploring the advantage of MSL. Besides, write atomicity is needed to achieve strong data consistency. This imposes requirements for the metadata size since NVM only supports 8-byte atomic updates. Thus, this section introduces bitmap-based metadata management.

MSL uses a per-node bitmap to indicate the validity of the shadow log. Each non-leaf node has two bits of metadata: the first bit (valid bit) is used to check whether the log of the current node is valid, and the other one (existing bit) indicates whether the descendant nodes have valid logs. To find the latest data, MSL first checks the root node's bitmap in the radix tree. If the existing bits are 0, indicating no valid shadow log beside the root node. Otherwise, MSL will search the child nodes and implement the following strategies according to the child nodes' bitmap:

Algorithm 1 Multi-granularity shadow logging

```
Require: Radix tree $Tree$, Operation $Op$, Offset $Off$, Length $Len$;
 1: procedure TRAVERSAL(lastnode, node, off, len)
 2:     if len == node.logSize or len $\leq$ MinLogSize then
 3:         if node.log is valid and Op == Write then
 4:             node = lastnode;
 5:         if node.log is not valid and Op == Read then
 6:             node = lastnode;
 7:         Access data from node;
 8:         return ;
 9:     if node.log is valid then
10:         lastnode = node;
11:     Get the start and end of child nodes with off and len
12:     for i = start to end do
13:         Get the newOff and newLen for childnode i
14:         TRAVERSAL(lastnode, node.child[i], newOff, newLen);
15:
16: TRAVERSAL($Tree.root, Tree.root, Off, Len$);
```

- If the valid bit is 1 and the existing bit is 0, all the latest data exists in the shadow log of this child node.
- If the valid bit is 0 and the existing bit is 1, MSL will search the next-level child nodes.
- If both the valid bit and the existing bit are 1, part of the latest data can be obtained from the child node, and other data is on the descendant node of this node.

There is no need to set the existing bit for leaf nodes with the smallest granularity because they do not have descendant nodes. Instead, MSL extends the valid bits of leaf nodes from one to several bits to support fine-grained updates. The size of the minimum update granularity depends on the number of valid bits. For example, if there are two valid bits in the leaf node and the granularity of the leaf node is 4KB, the finest granularity of updates would be 2KB. Each bit corresponds to half of the data in the leaf node. The latest data for the position where the existing bit is 1 exists in the shadow log of the leaf node, and the other latest data exists in the last ancestor node whose valid bit is 1. When a write operation comes, MSL first sets the existing bits on the traversal path if they are 0. Then MSL writes the data to the log if the valid bit is 0 and sets the valid bit; otherwise, MSL would clean the valid bit and write the data to the last valid ancestor node. If all the child nodes' valid and existing bits are 0, the existing bit of the node will be set to 0.

Figure 5 shows the bitmap-based metadata management of the file in Figure 4. The numbers in black show the initial bitmap state of this 32KB file (blue rectangles). The bitmap of the root node is `11`, the first bit (valid bit) is 1, meaning that

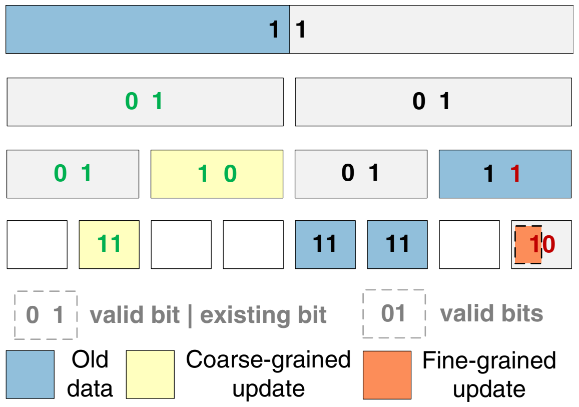

Fig. 5. Bitmap-based metadata management. The non-leaf node has valid and existing bits, whereas the leaf node only has valid bits.

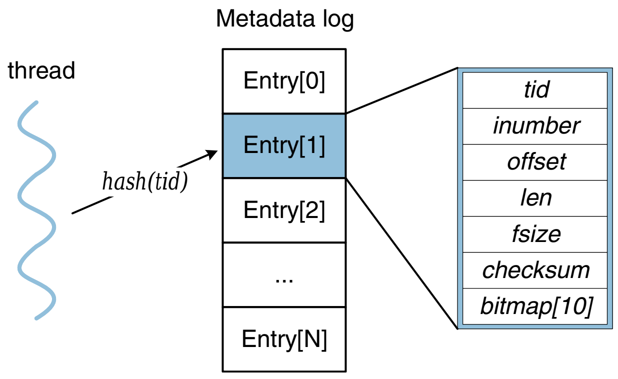

Fig. 6. Lock-free metadata log. MGSP uses hashing to implement lock-free metadata logging. Hash collisions are resolved with linear probing.

the log of the root node is valid, and the second bit (existing bit) is 1, meaning that a part of the latest data is stored in other nodes. This part of the data can be found in the right subtree because the existing bit of the right child node is 1. The `11` in the leaf node means this 4KB node is full of the latest data. The green and orange numbers represent bitmap changes during one coarse-grained update and one fine-grained update. For non-leaf node updates, only a valid bit will be set to indicate the location of this part of the data. For leaf node update, the bitmap will be set to `01`, `10`, or `11` based on the write size and offset.

Lazy cleaning for bitmap. For a coarse-grained update, the bitmap of all descendant nodes of the updated node should be nullified. To reduce the overhead, MSL only invalidates the bitmap of the updated node. The subsequent writes can invalidate the remaining bitmaps when writing this part of the data. In this way, the overhead of updating the entire subtree is spread over subsequent writes.

C. Crash Consistency Design

We propose lock-free metadata logging and multi-granularity locking to guarantee atomic write and data isolation.

1) Lock-free metadata log: Before the bitmap is modified, the shadow logging mechanism ensures that any data writing will not affect the consistency of the original file data. Therefore, we only need to ensure atomic modification of the bitmap to guarantee the atomicity of each write. To this end, MGSP designs a metadata log, as shown in Figure 6. The default size of a metadata log entry is 128 bytes, and each log entry consists of the necessary information, such as file size, I/O length, and I/O offset, to ensure the system's consistency under any crash. MGSP persists all necessary information into the metadata log before modifying the metadata.

The metadata log combines the operating log and undo log for bitmap. Only the valid bits must be recorded, and the existing bits can be recovered from the valid bits. To avoid lock competition caused by global metadata, MGSP uses hash and compare-and-swap (CAS) instructions to achieve lock-free metadata logging. Each thread can obtain a private log entry based on the hash of the thread ID. MGSP uses linear probing to deal with hash collisions while obtaining an empty log entry. This way, a 4KB metadata area can allow up to 32 threads to access files concurrently. For more than 32 threads, MGSP can expand the metadata area or wait for other threads to release log entries after the operation is completed. With the metadata log, MGSP can achieve operation-level atomicity.

For every write, the length of the range is limited. For example, on Linux, the size of each write will not exceed 2GB [28]. Therefore, we need logs with five different granularities (64B, 4K, 256K, 16M, 1G) at most to satisfy a single write with a radix tree in degree 64. The bitmap-based management could effectively reduce the metadata overhead of MSL because we need to update two nodes per level at most. So ten bitmap slots are enough for one write with arbitrary lengths on Linux.

Although MGSP reserves ten bitmap slots for each log entry, not every operation will use all slots, especially for aligned operations. For example, only one bitmap is required for a commonly aligned 4KB write. The multi-granularity feature can further reduce the number of required bitmaps, as the large-sized updates can use the bitmap of coarse-grained blocks without updating many fine-grained nodes. Therefore, MLSP does not need to flush all 128-bytes metadata logs. When the number of used bitmaps is less than 3, the first 64 bytes of the metadata log is enough. MGSP will only flush part of one metadata log entry to reduce the overhead of cache flushing. MGSP uses the checksum to ensure one metadata log is valid. After the update is finished, the length in the log will be set to 0 to mark it as outdated.

2) Multi-granularity data Lock:

Radix tree-based multi-granularity data management naturally supports Multiple Granularity Locking (MGL) [16]. MGL locking mechanism can lock the same data area with different granularities. In MGSP, we can lock the corresponding data area by locking a node. Nodes at different levels correspond to different locking granularities. For any lock range, MGL can lock the nodes of different levels to approximate the range locks, and the tree structure can quickly determine whether there are conflicting ranges. Using the combination of different granularity locks, MGL can reduce the overhead of locks as much as possible to improve parallelism. In addition

TABLE I
LOCK COMPATIBILITY TABLE FOR MULTIPLE GRANULARITY LOCKING.

	IR	IW	R	W
IR	✓	✓	✓	X
IW	✓	✓	X	X
R	✓	X	✓	X
W	X	X	X	X

to conventional read-write locks, MGL includes Intention Read (IR) and Intention Write (IW) locks. These two types of locks indicate that although a read/write lock does not lock the current range, a part of the range is locked by a more fine-grained read/write lock. Intention locks can avoid conflicts among different-granularity locks within the same range. Table I shows the compatibility between intention locks and read/write locks.

Before performing writes and reads, MGSP achieves isolation by locking the corresponding shadow logs with read-/write locks. MGSP finds the shadow logs according to the mechanism of MGL and applies IR/IW locks to the nodes along the way from the root node. The order of locking is according to the offset from small to large to avoid deadlock. The nodes will be unlocked in the same order when reads and writes are completed. MGL is designed for intra-process parallelism, a more common scenario than inter-process file sharing [8]. Currently, if one process wants to access a file opened by another via MGSP, it can only wait for the file to be closed. Other methods, such as pipes or shared memory, are more suitable than MGSP for applications that require frequent access to shared files between processes. Applications can choose whether to use MGSP to manage files according to their needs.

Lazy cleaning for intention lock. MGL needs to add intention locks to nodes along the search path and unlock after the operation is completed. Repeated lock-unlock operations on the critical path could degrade the performance. Because of that, MGSP also adopts the lazy cleaning strategy for intention locks. MGSP does not clean the intention locks during unlock stage. MGSP will try to obtain read/write locks on all child nodes when other locks conflict with intention locks. Lazy cleaning for intention lock will increase the overhead of coarse-grained locking. This optimization strategy is effective because the access size tends to be stable. The problem of increased overhead occurs only when the access size of the same data becomes different.

Greedy locking. In addition to lazy cleaning, MGSP can achieve a single locking strategy similar to range locks. It is enough to only lock the root node of the minimum search tree. This locking strategy locks data beyond the required range and is called greedy locking. To avoid the influence on scalability, MGSP may use greedy locking only when the current file has no more than one reference. When the bitmaps of the minimum search tree are all 0, it indicates that there may be no writing to the current subtree, and greedy locking can also be enabled for reading operations.

D. Write and Read Flows

MGSP aims to achieve more efficient crash-consistent memory-mapped IO, mainly to reduce the overhead of read and write operations. This section will show how to read and write data with MGSP.

Write. For writing a file, MGSP first hashes the thread ID to find the corresponding metadata log and writes the file size and other information to the log. Then MGSP tries the minimum search tree to avoid the whole tree search. If not hit, MGSP searches from the root node of the radix tree to find the correct minimum search tree according to the MSL and uses greedy locking if the conditions meet. MGSP modifies the existing bits on the search path and finds the corresponding data block. Before writing the data to the corresponding shadow log, MGSP locks the data block according to the MGL if greedy locking is not used and writes the modified valid bit to the metadata log. When all logs are written, the metadata checksum is calculated and written to the metadata log, and the metadata log is eventually persisted. Next, MGSP updates the valid bitmap of the updated node based on the metadata log and updates the minimum search tree if needed. Finally, MGSP unlocks all data blocks in the same order as locking.

Read. For reading a file, it first queries the minimum search tree and decides whether to use greedy locking. If the greedy locking is not applicable, the read locks of the corresponding data block are obtained according to the MGL. After the data is locked, the latest data is read according to the valid and existing bits. Finally, MGSP modifies the minimum search tree cache if needed and unlocks all data blocks in the same order.

Close and Recovery. When a file is no longer opened by any thread, MGSP will write all logs back to the original file and release related metadata. MGSP can use the metadata log to find and complete unfinished operations if a crash occurs and then write all the logs back. By comparing the bitmap saved in the metadata log with the actual bitmap, MGSP can complete the remaining metadata modification to complete the interrupted operation. We crash a random-write workload at random points and measure the time to recover the file from the logs. It takes 186ms to restore a 1GB file, of which 153ms is used to write a total of 48K log entries (189MB of data) back to the file. The number of logs that need to be replayed is related to specific workloads, but the total size will not exceed the file size. So even the worst case can be completed within 1s.

IV. EVALUATION

In this section, we present the implementation details and evaluation environment of MGSP and show the results on microbenchmarks and real-world applications.

A. Implementation and Experimental Setup

The proposed MGSP is implemented based on Libnvmmio. We implement the multi-granularity shadow log with metadata log and multi-granularity lock to replace the original logs in Libnvmmio and completely redesign the read-write flow. Like Libnvmmio, MGSP runs as a user library that uses

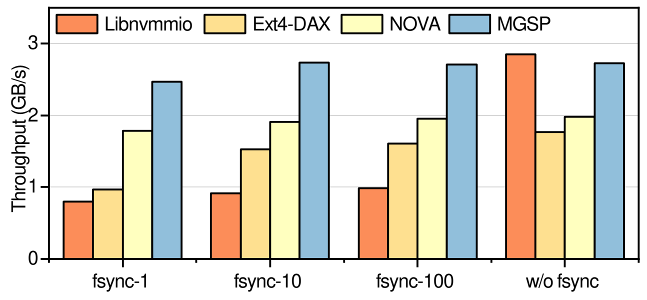

Fig. 7. Performance comparison with different sync intervals. *fsync-x* in the X-axis means one `fsync` is performed every *x* write operations.

`LD_PRELOAD` to intercept POSIX calls and the PMDK library to persist data to NVM. All the logs and metadata that need to be persisted are allocated from memory-mapped files. By utilizing well-designed data structures, MGSP avoids transitional locks in most cases and uses built-in atomic functions from GCC instead.

The experiment testbed is equipped with 2× Intel Xeon Gold 5317 CPUs, 128GB DRAM, and 512G Intel Optane DC Persistent Memory. The PM consists of four 128G Optane memory through interleaved mode. We compare the performance of MGSP with Ext4-DAX [27], Libnvmmio [9], and NOVA [41]. We use Ext4-DAX as the underlying file system for Libnvmmio and MGSP. They have different guarantees for data consistency, as shown below:

- **Ext4-DAX:** The Ext4 file system with direct access for files. It only supports metadata consistency.
- **Libnvmmio:** The latest work for crash-consistent MMIO. Libnvmmio adds the data consistency to the Ext4-DAX but only supports atomicity with fsync.
- **NOVA:** A kernel-space log-structured file system for NVM. The data atomicity is guaranteed for every operation with the CoW update.
- **MGSP:** Our solution. MGSP adds data consistency to the Ext4-DAX and supports operation-level atomicity. Each operation is a synchronized atomic operation.

For FIO, we run the experiments on a 1GB file for 60 seconds or after the performance is stable. Each experiment was run three times and averaged. Figure 7 shows the 4K sequential write performance comparison at different sync frequencies. The results show that although Libnvmmio can improve performance compared to Ext4-DAX without sync, the performance will drop significantly even if only one sync is performed every 100 writes. The performance of ext4-DAX also drops when each operation is synced. MGSP, on the other hand, hardly receives the impact of sync because each operation is already a synchronized atomic operation. For a fair comparison, the experiments below will sync for each operation.

B. Microbenchmark

MGSP can effectively improve the write performance by reducing redundant writes with shadow paging. Figure 8(a) and Figure 8(b) show the sequential and random write performance of Ext4-DAX, Libnvmmio, NOVA and MGSP at different granularities.

For fine-grained sequential write with a size less than 4KB, MGSP achieves $3.31 \sim 4.21\times$, $3.43 \sim 4.53\times$ and $1.69 \sim 2.06\times$ better performance than Ext4-DAX, Libnvmmio, and NOVA, respectively. The performance gain comes from the fine-grained update mechanism of MGSP, which reduces write amplification as much as possible. For writes with a size greater than or equal to 4KB, MGSP outperforms Libnvmmio by $3.23 \sim 4.3\times$, Ext4-DAX by $1.1 \sim 2.52\times$ and NOVA by $1.01 \sim 1.43\times$ for sequential write. Through the memory-mapped shadow log design, MGSP brings better performance by avoiding the complex software stack and the overhead of double writing. For random write, MGSP achieves $2.52 \sim 2.97\times$ and $2.56 \sim 3.16\times$ for fine-grained, $1.11 \sim 2.33\times$ and $2.72 \sim 3.46\times$ for coarse-grained than Ext4-DAX, Libnvmmio. Compared to NOVA, MGSP can achieve $1.15 \sim 1.4\times$ performance on fine-grained updates and is comparable to NOVA in most coarse-grained cases with a maximum performance loss of 17.9%.

While MGSP is not specifically designed for read, it can still improve read performance compared to Ext4-DAX due to the efficient MMIO and achieve comparable read performance to Libnvmmio. Figure 8(c) and Figure 8(d) show the sequential and random read performance of these file systems at different granularities. Compared with Ext4-DAX, MGSP can improve the read performance by $1.89 \sim 3.07\times$ on fine-grained and $1.26 \sim 1.33\times$ on coarse-grained sequential read. For random read, the improvement is $1.88 \sim 2.19\times$ and $1.28 \sim 1.71\times$. Compared with Libnvmmio, MGSP has similar coarse-grained sequential read performance and $11.3 \sim 14.1\%$ improvement on fine-grained read. For fine-grained random read, MGSP can improve the performance by $12.4\% \sim 21.6\%$ over Libnvmmio and has similar performance on coarse-grained read accesses. Another kernel-space file system, NOVA, is similar to Ext4-DAX in fine-grained reading but performs better in coarse-grained reading. MGSP has $1.46 \sim 2.18\times$ and $1.5 \sim 1.75\times$ improvement over NOVA on fine-grained sequential and random reads. Although the stronger consistency and improved write performance achieved by MGSP introduce additional overhead on reading operations, such as bitmap queries and fine-grained locking, the proposed optimization strategy can effectively reduce the overhead and improve performance.

Besides pure read/write operations, we also evaluate the performance under mixed read and write. We evaluate the 4KB access performance of Libnvmmio, NOVA, and MGSP under different write ratios, where the performance is normalized to Ext4-DAX. The results are shown in Figure 9. Although Libnvmmio can improve the throughput by 50.2% compared to Ext4-DAX with a read-write ratio of 9:1, the performance becomes worse than that of Ext4-DAX when the proportion of write operations reaches 50%. On the other hand, NOVA and MGSP can achieve a stable performance improvement of $58.7\% \sim 92.2\%$ and $113.1\% \sim 141.3\%$ under different write ratios, respectively. This proves that the synchronous write operation and shadow log mechanism of MGSP can

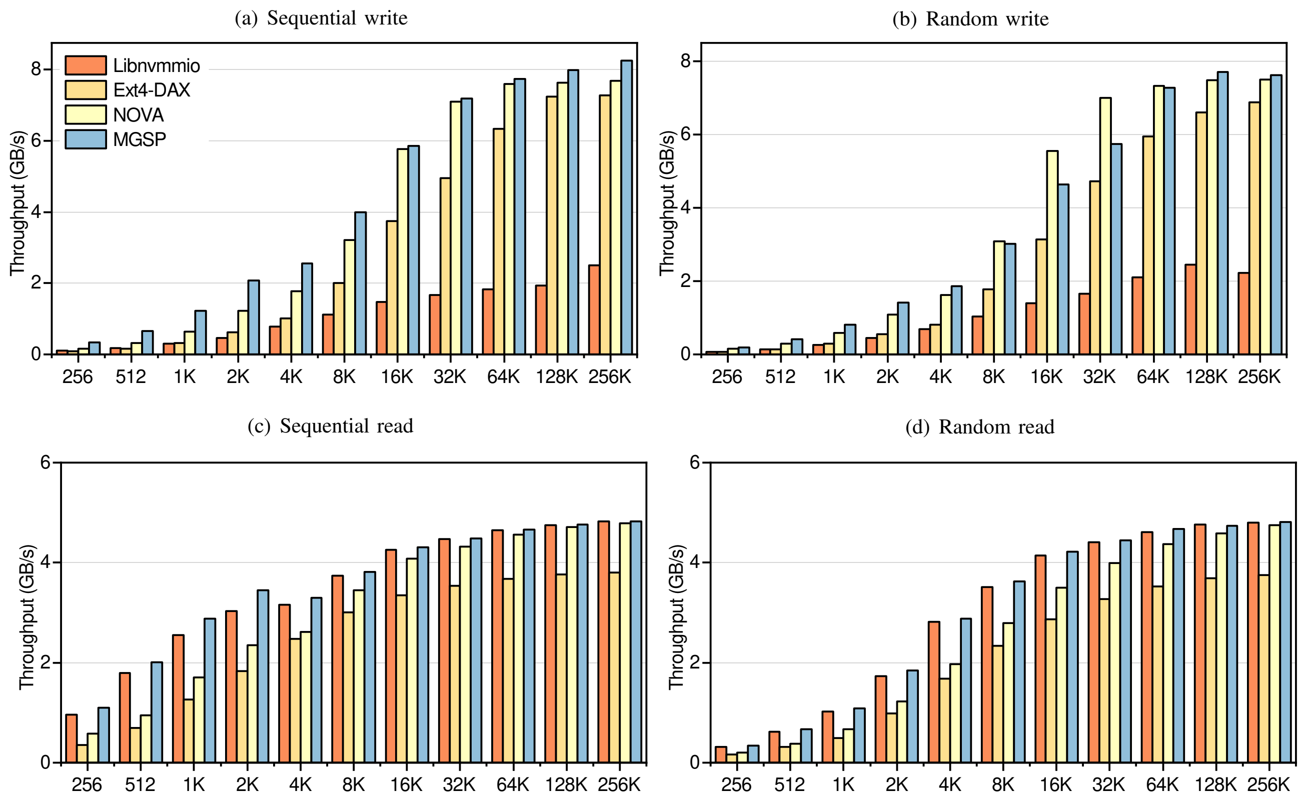

Fig. 8. Micro-benchmark.

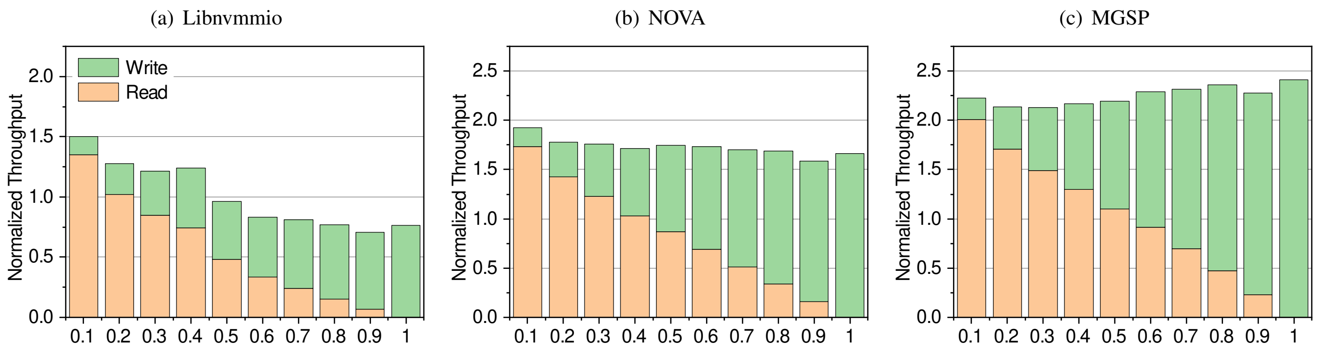

Fig. 9. Mixed read-write under different write ratio.

effectively avoid the influence of front and back thread conflict and redundant writes.

C. Scalability

We evaluate scalability with FIO's multi-threaded IO on the same file. The result is shown in Figure 10. Ext4-DAX and NOVA exhibit limited scalability. The conflict between the front and back threads makes Libnvmmio's performance not even increase with the number of threads. In all 1KB and 4KB cases, MGSP achieves the best scalability. In coarse-grained writing, scalability is primarily limited by hardware performance. On fine-grained writes, MGSP improves sequential writes by $3.81 \sim 8.51\times$, $3.14 \sim 57.6\times$ and $1.89 \sim 6.16\times$ compared to Ext4-DAX, Libnvmmio, and NOVA, respectively. Under 4KB granularity, the performance of MGSP is $2.56 \sim 3.76\times$ and $2.13 \sim 3.51\times$ that of Ext4-DAX for sequential and random access, respectively. For 16KB write, MGSP can achieve similar performance compared to Ext4-DAX and NOVA. These performance improvements mainly come from well-designed multi-granularity locking, which reduces lock overhead while ensuring parallelism.

D. Real Application

In this subsection, we evaluate the performance of MGSP on real applications with SQLite. SQLite is not designed for persistent memory, so there is no strong requirement for the synchronization of the underlying file system, and the consistency guarantee provided by MGSP cannot be fully utilized. Even so, MGSP can still bring performance benefits.

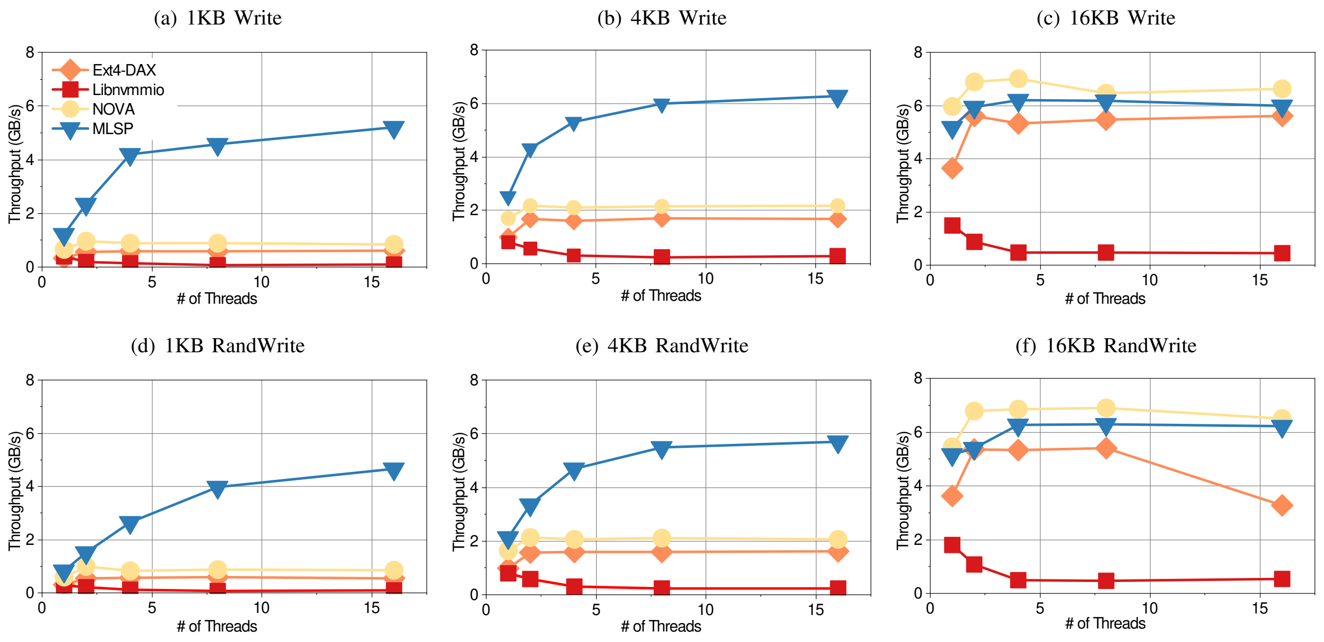

Fig. 10. Scalability comparison of different methods.

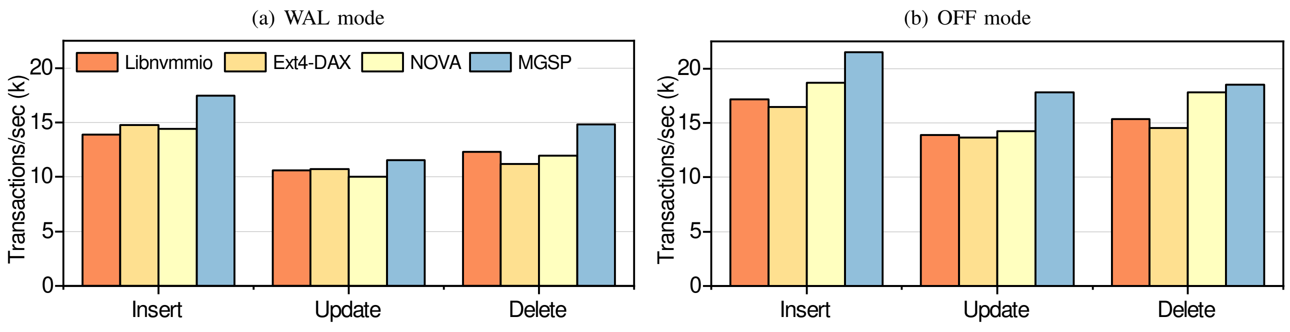

Fig. 11. SQLite with Mobibench.

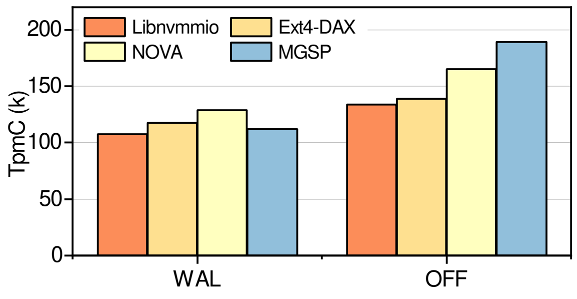

Fig. 12. SQLite with TPCC.

We use Mobibench to test the performance of basic transactions in SQLite's default WAL mode, as shown in Figure 11(a). Compared with Ext4-DAX, MGSP can improve the performance by 18.3%, 7.9%, and 32.5% on insert, update, and delete transactions, respectively. Compared with Libnvmmio, MGSP can improve the performance by 25.7%, 9.2%, and 20.6% on these three test modes. Considering that the file system can provide the required consistency guarantees, the logging mechanism of the database software itself will no longer be required [9], [21]. We also test the performance in OFF mode, as shown in Figure 11(b). In OFF mode, MGSP can improve the performance by 30.6%, 30.1%, and 27.6% on insert, update and delete compared to Ext4-DAX and provide consistency guarantees that Ext4-DAX does not support. Compared with Libnvmmio, MGSP can improve the performance by 25.3%, 28%, and 20.9%.

We also test the TPCC workload on SQLite. MGSP performs similarly with Ext4-DAX and Libnvmmio in WAL mode. In OFF mode, MGSP improves performance by 36.5%, 41.3%, and 14.6% compared to Ext4-DAX, Libnvmmio, and NOVA, respectively. It should be noted that although MGSP provides file-system-level atomicity, it does not have a transaction-level atomic mechanism. We hope to add related designs in future work so that existing database software can obtain corresponding performance gains without modification.

E. Write Amplification

MGSP and Libnvmmio use the PMDK library to write data and metadata in user space. Table II shows the ratio

TABLE II
AMPLIFICATION RATIO FOR RANDOM WRITE OF 10 SECONDS.

	Libnvmmio	Libnvmmio-100	Libnvmmio-wo-sync	MGSP
1K	2.048	1.997	1.061	1.088
4K	2.013	1.967	1.012	1.021
16K	2.002	1.956	1.001	1.014

of the write size received at the PMDK library to the write size at the file system layer. Libnvmmio avoids serious write amplification thanks to the differential logging during fine-grained updates. For example, it can write a portion of a 4KB data block when writing at 1KB granularity. However, executing sync forces Libnvmmio to write the log to the original file before the next write. Even if the frequency of sync is reduced, e.g., every 100 writes, the double-write problem still exists with a write amplification ratio of around 2. In contrast, MGSP can write only one copy of data and a small amount of metadata, effectively reducing write amplification and approaching the ideal result of Libnvmmio without sync (close to 1) while ensuring the atomicity of operations.

F. Performance Breakdown

Figure 13 shows the contributions of each technique to the final performance improvement. We give three examples with different thread counts and write granularities. In the example of 1KB write with a single thread, the performance of MGSP is $4.06\times$ that of Ext4-DAX. The main improvement comes from multi-granularity shadow logging. The shadow log can avoid writing the data twice to protect the data consistency, and the fine-grained logging further reduces write amplification. In the example of four-thread 4KB writing, fine-grained locking contributes the most, effectively improving threads' concurrency. Combined with other optimization techniques, a total performance improvement of $3.42\times$ is achieved. In the two-thread 2KB write case, fine-grained shadow logging and fine-grained locking contribute to the performance improvement, and the overall performance is $2.98\times$ compared to Ext4-DAX.

V. RELATED WORK

NVM-aware Filesystem. Ext4-DAX [27] bypasses the block layer and page cache for Ext4 to access the NVM directly. Ext4-DAX can only support atomicity for metadata, so it cannot ensure data consistency. BPFS [12] uses short-circuit shadow paging to provide fine-grained atomic updates to NVM and enforces durability and ordering with CPU cache management. PMFS [14] is a lightweight POSIX DAX file system and uses a hybrid consistency mechanism with in-place updates, fine-grained metadata logging, and CoW for data. NOVA [41] extends the log-structured technique to exploit NVM's fast random access characteristics for the hybrid memory system. NOVA supports strong data consistency and atomicity. Strata [23] proposes a hybrid file system that employs NVM as the first layer in user space to speed up I/O transmission with a log-structured approach. KucoFS [8] uses kernel-userspace collaboration to speed up metadata operations with CoW write and range lock.

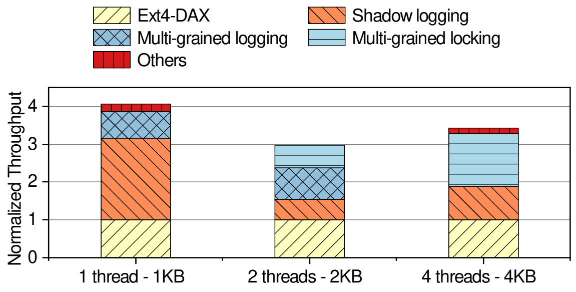

Fig. 13. Technique contributions for write performance.

Crash-consistent MMIO. SCMFS [40] uses mmap to map file data to virtual address space for lightweight and straightforward file access. However, it only supports metadata consistency. NOVA [41] uses Copy-on-Write to support data consistency with atomic-mmap, which will cause write amplification and lose the advantages of MMIO. SplitFS [21] uses mmap files to support user-level IO with flexible crash-consistency guarantees. It uses relink to achieve atomicity but still needs copy-on-write to ensure data consistency in strict mode. Libnvmmio [9] is a user-level failure-atomic memory-mapped interface. Libnvmmio uses efficient hybrid logging for data consistency but not operation-level atomicity.

Bitmap-based Failure-Atomic. Shadow Sub-Paging (SSP) [31] is also a failure-atomic mechanism using a bitmap to remap the data. The atomicity is preserved with metadata journaling. However, SSP aims to eliminate the redundant writes of persistent memory transactions. SSP only supports cache-line-level remapping between two data blocks of the same size to meet the transaction requirements. On the contrary, MGSP aims to achieve crash-consistent memory-mapped IO for file systems. MGSP designed the tree-based multiple-granularity shadow log to handle fine-grained and coarse-grained access.

VI. CONCLUSION

We present MGSP, a crash-consistent mechanism for efficient memory-mapped I/O on NVM. MGSP manages the file data based on the idea of shadow logging. MGSP minimizes the redundant data copies in the crash-consistent mechanism by remapping the file data to different granularity logs. High scalability is achieved with atomic function-based multiple-granularity locking. Experimental results show that the proposed MGSP achieves $1.1 \sim 4.21\times$ write performance improvement and 36.5% improvement for TPCC on SQLite.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their valuable comments. This work was supported in part by the Research Grants Council of the Hong Kong Special Administrative Region, China under Grant CityU 11209122, and by the National Natural Science Foundation of China under Grant No. 62202396. The corresponding author is Qiao Li (qiaoli045@gmail.com).

REFERENCES

[1] "3d xpoint: A breakthrough in non-volatile memory technology."

[2] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *Proceedings of the 18th Annual International Conference on Supercomputing*, ser. ICS '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 277–286.

[3] N. Amit, "Optimizing the TLB shootdown algorithm with page access tracking," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 27–39.

[4] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Implications of cpu caching on byte-addressable non-volatile memory programming," *Hewlett-Packard, Tech. Rep. HPL-2012-236*, 2012.

[5] M. S. Bhaskaran, J. Xu, and S. Swanson, "Bankshot: Caching slow storage in fast non-volatile memory," *SIGOPS Oper. Syst. Rev.*, vol. 48, no. 1, p. 73–81, may 2014. [Online]. Available: https://doi.org/10.1145/2626401.2626417

[6] A. B. Bityutskiy, "Jffs3 design issues," 2005.

[7] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing safe, user space access to fast, solid state disks," *SIGPLAN Not.*, vol. 47, no. 4, p. 387–400, mar 2012.

[8] 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)*. USENIX Association, Feb. 2021, pp. 81–95.

[9] J. Choi, J. Hong, Y. Kwon, and H. Han, "Libnvmmio: Reconstructing software IO path with Failure-Atomic Memory-Mapped interface," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 1–16.

[10] J. Choi, J. Kim, and H. Han, "Efficient memory mapped file I/O for In-Memory file systems," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. Santa Clara, CA: USENIX Association, Jul. 2017.

[11] Y. Choi, I. Song, M.-H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo, J. Shin, Y. Rho, C. Lee, M. G. Kang, J. Lee, Y. Kwon, S. Kim, J. Kim, Y.-J. Lee, Q. Wang, S. Cha, S. Ahn, H. Horii, J. Lee, K. Kim, H. Joo, K. Lee, Y.-T. Lee, J. Yoo, and G. Jeong, "A 20nm 1.8v 8gb pram with 40mb/s program bandwidth," in *2012 IEEE International Solid-State Circuits Conference*, 2012, pp. 46–48.

[12] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 133–146.

[13] J. Corbet, "Supporting filesystems in persistent memory."

[14] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: Association for Computing Machinery, 2014.

[15] R. Gioiosa, J. Sancho, S. Jiang, and F. Petrini, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005, pp. 9–9.

[16] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, *Granularity of Locks and Degrees of Consistency in a Shared Data Base*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988, p. 94–121.

[17] R. Hagmann, "Reimplementing the cedar file system using logging and group commit," *SIGOPS Oper. Syst. Rev.*, vol. 21, no. 5, p. 155–162, nov 1987.

[18] D. Hitz, M. Malcolm, and J. Lau, "File system design for an NFS file server appliance," in *USENIX Winter 1994 Technical Conference (USENIX Winter 1994 Technical Conference)*. San Francisco, CA: USENIX Association, Jan. 1994.

[19] "Intel architecture instruction set extensions programming reference."

[20] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. S. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane DC persistent memory module," *CoRR*, vol. abs/1903.05714, 2019. [Online]. Available: http://arxiv.org/abs/1903.05714

[21] 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*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 494–508.

[22] t. kawahara, "Scalable spin-transfer torque ram technology for normally-off computing," *IEEE Des. Test*, vol. 28, no. 1, p. 52–63, jan 2011.

[23] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 460–477.

[24] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 2–13. [Online]. Available: https://doi.org/10.1145/1555754.1555758

[25] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," *SIGPLAN Not.*, vol. 31, no. 9, p. 84–92, sep 1996.

[26] G. Lee, W. Jin, W. Song, J. Gong, J. Bae, T. J. Ham, J. W. Lee, and J. Jeong, "A case for hardware-based demand paging," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 1103–1116.

[27] Linux, "Direct access for files," 2011.

[28] "write(2) — linux manual page."

[29] A. Memaripour and S. Swanson, "Breeze: User-level access to non-volatile main memories for legacy software," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018, pp. 413–422.

[30] C. Mohan, "Repeating history beyond aries," in *Proceedings of the 25th International Conference on Very Large Data Bases*, ser. VLDB '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, p. 1–17.

[31] Y. Ni, J. Zhao, H. Litz, D. Bittman, and E. L. Miller, "Ssp: Eliminating redundant writes in failure-atomic nvrams via shadow sub-paging," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 836–848.

[32] D. Park and D. Shin, "Ijournaling: Fine-grained journaling for improving the latency of fsync system call," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '17. USA: USENIX Association, 2017, p. 787–798.

[33] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 8.

[34] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

[35] N. Y. Song, Y. Son, H. Han, and H. Y. Yeom, "Efficient memory-mapped i/o on fast storage device," *ACM Trans. Storage*, vol. 12, no. 4, may 2016.

[36] M. M. Swift, "Towards o(1) memory," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 7–11. [Online]. Available: https://doi.org/10.1145/3102980.3102982

[37] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," in *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2011, pp. 340–349.

[38] D. Vučinić, Q. Wang, C. Guyot, R. Mateescu, F. Blagojević, L. Franca-Neto, D. L. Moal, T. Bunker, J. Xu, S. Swanson, and Z. Bandić, "DC express: Shortest latency protocol for reading phase change memory over PCI express," in *12th USENIX Conference on File and Storage Technologies (FAST 14)*. Santa Clara, CA: USENIX Association, Feb. 2014, pp. 309–315.

[39] M. Wilcox, "Add support for nv-dimms to ext4."

[40] X. Wu and A. L. N. Reddy, "Scmfs: A file system for storage class memory," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–11.

[41] 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). Santa Clara, CA: USENIX Association, Feb. 2016, pp. 323–338.

[42] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, "Nova-fortis: A fault-tolerant non-volatile main memory file system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 478–496.

[43] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt." in *FAST*, vol. 12, 2012, pp. 3–3.

[44] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *FAST*, 2015.

[45] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015, pp. 1–10.

[46] R. Zwisler, "Add support for new persistent memory instructions."

Appendix

A. Abstract

Our artifact provides the source code of the proposed MGSP, the baseline Libnvmmio, and the benchmark FIO, along with scripts to compile the code and set up the underlying file system. We also provide the scripts to regenerate all the results of FIO in this paper.

B. Artifact check-list (meta-information)

- **Algorithm:** Multi-Granularity Shadow Paging (MGSP).
- **Program:** FIO.
- **Compilation:** gcc 7.5.0 or above.
- **Hardware:** Intel Optane DC Persistent Memory (PM) and the supported processor. (Not needed if simulating PM with DRAM, the conclusion is the same, but the results might be different.)
- **Metrics:** Throughput.
- **Output:** The throughput result files.
- **How much time is needed to complete experiments (approximately)?:** About three hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache License 2.0.

C. Description

1) How to access: Our source code, benchmark and scripts are available on Github: `https://github.com/MIoTLab/MGSP`. You can find the latest information in README.md.

2) Hardware dependencies: We recommend testing on Inter Optane DC Persistent Memory 200 Series and 3rd Generation Intel Xeon Processors. Four 128 GB Intel Optane PM and Intel Xeon Gold 5317 is our test system, but similar results can be obtained with other Optane PM models and matching processors. If the hardware requirements cannot be met, DRAM can be used to simulate the PM, which can also prove the effectiveness of our method, but the resulting numbers might be different.

3) Software dependencies: We use *make and* gcc to compile the source code. The PMDK library is needed to write data to PM. The OS of our test system is Ubuntu 18.04 with Linux kernel 5.1.0. We choose this setting for a fair comparison with NOVA.

D. Installation

To install PMDK on Ubuntu 18.04 or above:

```
$ sudo apt install libpmem-dev
```

Install other dependencies:

```
$ sudo apt install git gcc make numactl
```

You can get our artifact from Github:

```
$ git clone https://github.com/MIoTLab/MGSP.git
```

Compile the benchmark and source code:

```
$ cd evaluation/fio/src
$ ./configure && make
$ cd ../../libnvmmio/src && make
$ cd ../../../src && make
```

Set up the underlying file system (Ext4-DAX or NOVA):

```
$ cd evaluation
$ sudo ./ext4_config.sh
or
$ sudo ./nova_config.sh
```

Before setting up NOVA, you must compile and install the Linux kernel with NOVA supported; refer to `https://github.com/NVSL/linux-nova`. Compiling the kernel is only necessary for NOVA results.

E. Experiment workflow

After setting up the file system, navigate to `evaluation/fio/scripts`. Just run this script to run all the evaluations:

```
$ cd evaluation/fio/scripts
$ sudo ./run_all.sh
```

The `run_all.sh` will run three scripts individually. To run the basic read/write experiment, run `sudo ./micro.sh`; it should take around 90 minutes. To run the mixed read-write experiment, run `sudo ./mix.sh` for around 20 minutes. To run the multi-threads experiment, run `sudo ./mul-thread.sh` for around 60 minutes.

These scripts will compile the source code and generate all the results besides NOVA. To get the results of NOVA, run the same scripts with `NOVA` attached, like `sudo ./micro.sh NOVA` after setting up the NOVA file system.

F. Expected results

After running the scripts, the result files will be output at `evaluation/fio/result`. Example output is given under `evaluation/fio/result/example_result`.

G. Experiment customization

All scripts are customizable to allow different parameters, like operations, file sizes, etc. We also provide a `run.sh` script for finer parameter settings. All other scripts are wrappers around it to run different experiments. It can be run like this:

```
$ sudo ./run.sh MGSP write 1g 4k 1 1 0 10 50
```

The meaning of these parameters are:

```
run.sh fs op fsize bs fsync t_num write_ratio
    runtime ramptime
  fs: the file system
  op: the operation
  fsize: the file size
  bs: the block size
  fsync: write number between each fsync
  t_num: the number of threads
  write_ratio: the write ratio for mixed operations
  runtime: the run time (in second)
  ramptime: the ramp time (in second)
```

Like Libnvmmio, MGSP uses `LD_PRELOAD` to intercept the system calls during runtime, and only the files opened with flag `O_ATOMIC` will be operated using this user-space library. The FIO benchmark has been modified with the flag. Other applications can use MGSP in the same way. First add the `O_ATOMIC` flag in file open operation, then compile and run like this:

```
$ sudo LD_PRELOAD=MGSP/src/mgsp.so ./your_app
```