Atomic but Lazy Updating with Memory-mapped Files for Persistent Memory

Qisheng Jiang, Lei Jia, and Chundong Wang*
School of Information Science and Technology, ShanghaiTech University,
Shanghai Engineering Research Center of Energy Efficient and Custom AI IC, Shanghai, China

***Abstract*—Applications memory-map file data stored in the persistent memory and expect both high performance and failure atomicity. State-of-the-art NOVA and Libnvmmio guarantee failure atomicity but yield inferior performance. They enforce data staying fresh and intact at the mapped addresses by continually updating the data there, thereby incurring severe write amplifications. They also lack the adaptability to dynamic workloads and entail housekeeping overheads with complex designs. We hence propose Acumen with a group of reflection pages managed for a mapped file. Using a simplistic bitmap to track fine-grained data slices, Acumen makes a reflection page and a mapped file page pair to alternately carry updates to achieve failure atomicity. Only on receiving a read request will it deploy valid data from reflection pages into target mapped file pages. The cost of deployment is amortized over subsequent read requests. Experiments show that Acumen significantly outperforms NOVA and Libnvmmio with consistently higher performance in serving a variety of workloads.**

***Index Terms*—Memory-map, Persistent Memory, Atomicity**

I. Introduction

Persistent memory (pmem), with byte-addressability, persistency, and fast access speed, provides a promising storage device to support applications. Manufacturers have shipped pmem products in NVDIMM [1]–[4] or non-volatile memory (NVM). Researchers have built new file systems for applications to explore the potential of pmem in the conventional form of files [5]–[7]. They mainly utilize the direct access (DAX) feature that bypasses DRAM page cache for CPU to store and load file data with pmem. However, the software overhead of using `write` and `read` system calls for file operations is non-trivial, due to the heavy traversal through multiple software layers across the user and kernel spaces [7, 8]. It is more efficient to memory-map file data into a contiguous memory space with `mmap` and `munmap`. Applications use `memcpy` to load and store file data mapped in the user's memory space at runtime, with evidently reduced software overhead.

With either `write` or `mmap`, applications need the underlying system software to enable the *failure atomicity* (all-or-nothing done) to correctly recover data to a consistent state upon a crash, e.g., a power outage or kernel panic. To avoid destroying the original copy for a successful recovery, the system software makes a backup copy for an in-pmem file page to be updated by copy-on-write (CoW) or logging [7]–[10]. Before mapping a file page, NOVA creates a replica page and copies data into the page [6]. It maps the replica page instead of the original one to receive updates. On a sync request (`msync` or `fsync`), NOVA copies newer data in the replica page back to overwrite the file page. Later, Libnvmmio was proposed to use hybrid logging for failure atomicity before updating mapped file pages [8]. Libnvmmio counts read and write requests in the past epoch framed by two consecutive syncs. It adopts undo logging if the ratio of read requests has reached a threshold (e.g., 40%). For a more write-intensive workload, it chooses redo logging.

Despite achieving failure atomicity, NOVA and Libnvmmio yield inferior performance. Both of them cause concrete write amplifications, as they abide by a de facto principle that systems must eagerly keep data up-to-date and intact for applications to access. Note that, after mapping file pages, applications can only access data at the mapped addresses in the memory-mapped I/O path. NOVA copies all data elsewhere to serve current memory-mapped I/Os and copies back for future use. Libnvmmio, in its either undo or redo logging mode, updates data at the mapped addresses after logging for the data. However, *data an application freshly writes needs not be ready in place until the application is to read it.* As long as an application receives correct and genuine data after issuing a read request, the place where data has been held is unimportant.

NOVA and Libnvmmio also lack the adaptability to dynamic workloads. NOVA always copies mapped data out and back. Libnvmmio's runtime switch on logging modes yet depends on its knowledge obtained in the past epoch. It may inefficiently handle I/O requests in the current epoch due to a lag effect. Worse, Libnvmmio initiates a switch on a sync request. It might remain ineffectual if applications have a low use of syncs in a long run. Moreover, Libnvmmio employs a complex multi-level indexing structure to index log entries for mapped pages, which incurs substantial housekeeping and traversal costs.

These observations motivate us to develop **Acumen** to help applications gain both high performance and failure atomicity with in-pmem data through a simplistic but effectual design.

- Acumen dedicates an in-pmem *reflection page* to a mapped file page that is receiving data updates. To serve fine-grained sub-page updates, it further partitions a page into slices, with a bitmap to track the validity of each slice in the file page.
- Acumen writes updated data alternately into either page without destroying the last valid copy. On a read request, if the target data stays valid in the reflection page, it deploys the data into the mapped file page for applications to access with the mapped address.

Acumen functions as a library with user-friendly interfaces in the user space, unbound to any particular file system. It writes data only once without eager in-place updating, and lazily

* C. Wang is the corresponding author (cd_wang@outlook.com).

978-3-9819263-7-8/DATE23/© 2023 EDAA

triggers a deployment only when an application is to read data that is invalid in the mapped file page. The cost of deployment is amortized with subsequent read requests. We implement and evaluate Acumen with micro- and macro-benchmarks. Acumen significantly outperforms NOVA and Libnvmmio. Take a write-intensive workload for example. The throughput of Acumen is 18.9× and 3.2× that of NOVA and Libnvmmio, respectively.

II. Background

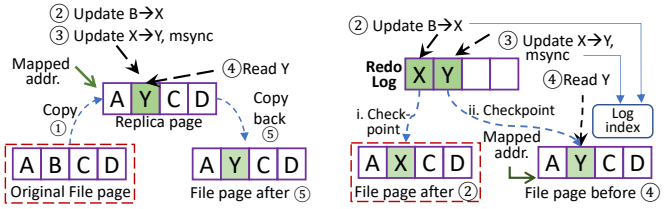
Pmem. In recent years, computer architects have used NVDIMM (DRAM backed by flash memory) [1]–[4] or NVM technologies (e.g., STT-RAM [11] and Intel Optane memory [12]) to make persistent memory (pmem) products to embrace both DRAM's byte-addressability and disk's persistency. New file systems have been built on top of pmem with the direct access (DAX) feature by which applications directly load and store data without buffering in DRAM page cache [5]–[7]. Though, using `write` and `read` to operate with in-pmem file data is time-consuming as they frequently switch between user and kernel spaces and traverse multiple software layers. In order to reduce software overheads, applications can memory-map a file and call `memcpy` to load and store data with pmem.

Atomic memory-mapped update. Mapping file data stored in pmem matches the byte-addressability of pmem. The persistency of pmem retains data being update after a system crash, but the crash may leave the update ambiguously half-done if no failure atomicity is guaranteed. To keep mapped data failure-atomic is hence a necessity for applications [5]–[10]. State-of-the-art NOVA and Libnvmmio utilize CoW or logging to do so [6, 8]. NOVA [6] duplicates file data in replica pages allocated elsewhere prior to mapping and updates data in them. On the call of `msync` or `munmap`, NOVA copies data back to refresh original pages. Libnvmmio [8] is a library that employs hybrid logging. It manages a log per mapped file, in which a log entry is indexed in a multi-level structure with the in-file offset as index key. Undo logging records original data in the log while redo logging logs updated data. Libnvmmio chooses undo or redo logging according to the counts of read and write requests in the past epoch framed by two consecutive sync requests. It uses undo logging by default and transits to redo logging if writes have taken no less than 40% in all requests.

III. Motivational Analysis

The intention of providing memory-mapped I/Os with the guarantee of failure atomicity is to exploit pmem's fast access speed, persistency, and byte-addressability. However, the designs of NOVA and Libnvmmio are ineffectual because of severe performance overheads in following dimensions.

Firstly, NOVA and Libnvmmio always keep the latest valid data intact and ready at mapped addresses, due to a de facto assumption that applications are to access the data at any time. Whereas, this entails excessive writes when they simultaneously guarantee the failure atomicity. Fig. 1 illustrates a mapped page composed of four data items on which an application consecutively modifies one item twice and then reads it. As shown in Fig. 1a, NOVA copies original file data into a replica page (①) and maps the replica page to serve write and read



(a) NOVA's CoW (b) Libnvmmio's Redo Logging

Fig. 1: A Comparison on CoW and Redo Logging

requests (②③④). On the `msync` request, it copies all data with updates back to the file (⑤). Libnvmmio logs original data with undo logging by default and puts updated data at the mapped addresses, writing the same volume of data twice on the critical path. It logs updated data with redo logging and checkpoints the data into mapped addresses, regardless of whether the data is truly to be accessed or not. As the example in Fig. 1 is more write-intensive, Libnvmmio adopts redo logging. In Fig. 1b, after each write request (②③) Libnvmmio checkpoints updated data to the file page, but the application only reads the second version (④). Therefore, the first checkpoint is unnecessary.

Secondly, NOVA and Libnvmmio lack the adaptability and flexibility in serving dynamic workloads. NOVA's CoW strategy always copies mapped pages out and back. Libnvmmio alternately configures undo or redo logging for read- and write-intensive workloads, respectively, but triggers a possible switch on encountering an explicit sync request. This is obviously inappropriate for workloads without a frequent use of syncs. Worse, Libnvmmio chooses to use redo or undo logging by counting historical write and read requests, which may not align with or even become against ongoing requests. Assuming that an application alternately switches between being write- and read-intensive after every sync request while Libnvmmio initially sets the undo logging as its default, it shall severely suffer from the lag effect and keep yielding inferior performance.

Thirdly, although Libnvmmio's logging is more efficient than CoW, the way it manages a log entry per file page causes non-trivial overheads. To locate a log entry, Libnvmmio partitions the in-file offset into five segments and traverses a hierarchical multi-level indexing structure. With either redo or undo logging, every write request necessitates a traversal to create a new logging record (see Fig. 1b). With redo logging, for every read request, Libnvmmio checks if the latest valid data exists in the log entry or not, so as not to read out stale data. The cost of housekeeping and traversal thus aggregates on almost every request and badly impairs performance.

IV. Design of Acumen

We propose **Acumen** that performs atomic but lazy updates with memory-mapped file data for persistent memory. Acumen manages a reflection page for each mapped file page and makes a one-to-one staging between them. It alternately writes either page for failure atomicity but lazily deploys the latest valid data into the mapped file page only on receiving a read request.

A. Acumen's Atomic `ato_memcpy` Interface

To differentiate ordinary `memcpy` from the atomic `memcpy` of Acumen, we introduce a new `ato_memcpy` which copies

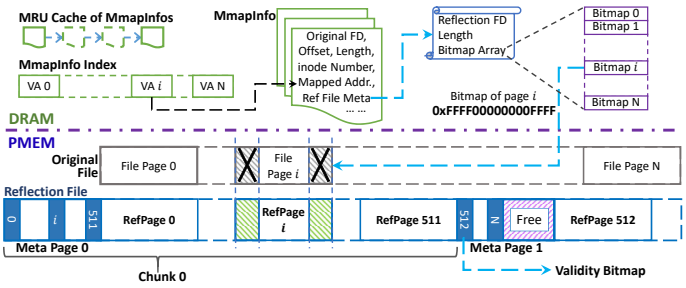


Fig. 2: An Overview of Acumen's Components.

data from/to file pages that are mapped into memory with the guarantee of failure atomicity. We also have `ato_mmap` and `ato_munmap` that initiate and close the mapping between in-pmem file pages and memory space, respectively.

B. Acumen's in-DRAM and Pmem Structures

In order to gain high efficiency in housekeeping with the fast pmem, Acumen incorporates minimal metadata for management. Fig. 2 shows its structures in pmem and DRAM. Acumen only stores data and metadata that are essential for failure atomicity in pmem while placing some metadata in DRAM to accelerate indexing and searching.

In-pmem structures. For a mapped file, Acumen creates a staging *reflection file* in pmem to absorb updates and preserve failure atomicity. The filename of reflection file is the `inode` number of original file, so Acumen leverages the underlying file system to implicitly record two files' pairing relation in a directory entry. Acumen memory-maps both files. A reflection page (RefPage) is paired to a mapped file page. With regard to a sub-page update that modifies data less than a page, it partitions a page into *slices* in accordance with the size of CPU cache line. Given the typical page size and cache line size of 4KB and 64B, respectively, a page holds 64 ($\frac{4KB}{64B}$) slices. To track if a slice is valid in the mapped file page or the RefPage, Acumen maintains a *validity bitmap* in 64 bits (8B) for each mapped page. The bitmap is the core metadata of Acumen. Acumen collects 512 ($\frac{4KB}{8B}$) such bitmaps in a *meta page* in the reflection file. As shown in Fig. 2, a reflection file is composed of multiple chunks and in one chunk there is a meta page and 512 RefPages. Given an offset λ_o in the original file, we can calculate its location λ_r in the reflection file as

$$\lambda_r = \lambda_o + \left\lfloor \left\lfloor \frac{\lambda_o}{4096} \right\rfloor /512 + 1 \right\rfloor \times 4096 \qquad (1)$$

In-DRAM structures. As shown in Fig. 2, Acumen manages informational metadata in DRAM with a structure named *MmapInfo* for a mapped file. An MmapInfo contains multiple parts, such as the original file's `inode` number, descriptor, length, mapped address, and offset as well as the reflection file's metadata like bitmaps. To support simultaneously mapping multiple files, Acumen builds an index structure for MmapInfos with the mapped addresses of original files as index keys.

C. Acumen's Write and Read Procedures

Acumen decouples write from read. It writes data only once without losing any failure atomicity. Only on receiving a read request raised by applications will it lazily deploy the latest data at the mapped addresses if that data stays valid in a RefPage.

Memory-Map. Acumen defines that there is at most one reflection file per mapped file even when an application memory-maps the same file for multiple times. Given a mapping request, Acumen firstly checks if a reflection file exists with the mapped file's `inode` number as the filename. If not, Acumen creates a new one and configures its size according to Eq. (1). Then, Acumen maps the reflection file into the application's memory space. Acumen next maps the original file, initializes metadata, and sets up the file's MmapInfo with an index inserted.

Write. When a write request arrives in `ato_memcpy`, Acumen handles it in the following steps. Firstly, it searches among MmapInfos and eventually fetches the target page's bitmap. As real-world applications continually issue a mixed stream of full- and sub-page updates, Acumen needs the 64-bit bitmap to indicate whether the data to be modified stays valid in the slices of original page ('0') or RefPage ('1'). For each involved slice, Acumen avoids overwriting the valid copy but writes newer data into the other file holding the invalid copy. For example, the bitmap of Page i in Fig. 2 is 0xFFFF00000000FFFF, so the valid data of slices 16 to 47 resides in the original page while the other data is valid in the RefPage i. On updating, for example, slices 0 to 15, Acumen writes the newer data into the original page. After writing all slices for the write request, it modifies in-DRAM and pmem bitmaps for the page. It also applies a memory fence (e.g., `sfence`) to enforce a persist order between writing data and updating the in-pmem bitmap and uses the change of in-pmem bitmap in an 8B atomic write as the end of `ato_memcpy`.

Read. When a user raises a read request via `ato_memcpy`, Acumen firstly fetches the bitmap of target page. If the bits for involved slices are all '0's, Acumen loads data from the original page like an ordinary `memcpy`. Otherwise, it triggers a deployment. Acumen deploys valid data from the RefPage to file page and then reads data at the mapped address.

(i) Firstly, Acumen checks the bitmap to find out what slices should be deployed, i.e., ones with '1's in the bitmap.
(ii) Acumen copies data in such slices from the RefPage to original file page.
(iii) Next, Acumen sets '0's for deployed slices in DRAM's bitmap and persistently updates the in-NVM bitmap using an 8B atomic write ordered by `sfences`.

Acumen lazily triggers a deployment only when it receives the first read request on a mapped page. Subsequent read requests on the same page amortize the cost of deployment.

Memory-Unmap. When receiving `ato_munmap` to close a mapping, Acumen provides two configurable unmap modes. In the *fast* mode, it closes both files and frees in-DRAM structures. On the next `ato_mmap`, Acumen reuses the same reflection file and rebuilds in-DRAM structures. In the *spatial* mode, Acumen deploys valid data in RefPages and then removes the reflection file. As a result, the fast mode is faster on exiting and retains the reflection file. The spatial mode saves pmem space but demands a new reflection file to be allocated on the next mapping.

D. Acumen's Failure Atomicity

Acumen uses memory fences to retain a persist order between modifying data and the in-pmem bitmap upon an update or deployment. On platforms without the eADR support, Acumen uses cache line flush instructions (e.g., `clwb`) to manually flush data and metadata in volatile CPU cache to pmem, while the eADR guarantees to automatically flush all CPU cache lines to pmem on a power outage [13]. If an application's write or read request involves a single page, Acumen adopts the atomic change of in-pmem bitmap to mark the success of writing or deploying slices. If a crash happens before the atomic write, Acumen regards the latest update failed and the unmodified in-pmem bitmap still records where valid data stays. If a failure occurs after changing the in-pmem bitmap, the up-to-date bitmap has already tracked the newest updates that Acumen renders retrievable for users. When applications do with multiple pages in one request, Acumen uses hardware transactional memory (HTM) to atomically change corresponding bitmaps [5]. To sum up, Acumen secures the failure atomicity and incurs no ambiguity in updating or reading memory-mapped file data.

E. Implementation for Acumen

To avoid wasting pmem space, Acumen reserves and adjusts the size of reflection file through the `ftruncate`[1] that does not truly allocate space until data is written down. Acumen also calls `fallocate`[2] to deallocate and reclaim an unused in-pmem reflection page when all bits in the page's bitmap are '0's, since all valid data is stored in the original page.

Acumen utilizes fine-grained read-write locks to share in-DRAM structures. For acceleration, it buffers the most-recently-used (MRU) MmapInfos in an in-DRAM cache (see Fig. 2). Acumen adopts the red-black tree to index all Mmapinfos while it uses a linked list to manage the Mmapinfo cache.

F. Optimization and Limitation

Acumen atomically handle write requests with memory-mapped files and efficiently serve read requests. Its reduced writes benefit both performance and lifetime for pmem [14, 15]. Yet there exist few special cases that shall be taken into account.

Sparse updating. Application may sparsely update data at distant locations which, for example, are much greater than 512 pages in the mapped file. As a result, an allocation of contiguous pages for the reflection file underutilizes pmem space. To deal with sparse updates, Acumen adds an extra option to format the meta page, in which alongside a file page's bitmap it places the page's number. Thus, a meta page compactly indexes 256 ($\frac{4KB}{16B}$) RefPages. Acumen accordingly adjusts in-DRAM structures to suit and index new meta pages.

Sharing between multi-processes. Users may employ Acumen to concurrently update data with the same mapped file across multiple processes. They can share the original file and reflection file with virtual file system (VFS) and underlying file system through locking/unlocking. But how to concurrently operate with in-DRAM structures is non-trivial, since each process has its private virtual memory space in DRAM. To handle this issue of inter-process sharing, Acumen makes use of POSIX shared memory with `shm_open`[3] and `mmap` to house and share the in-DRAM structures for concurrent accesses.

Viability. Acumen is implemented as a library in user space, not bound to any particular underlying file system. Modifying applications from `memcpy` to Acumen's `ato_memcpy` is not difficult. For example, we just change 14 lines of code for SQLite [16] to use Acumen's interfaces. Although Acumen is designed with the context of memory-mapped I/Os, its idea can be applied to reshape `write` and `read` system calls [7, 8].

Mixed use. Like NOVA and Libnvmmio, a mixed use of `ato_memcpy` with ordinary `memcpy` or `write` is disallowed, as that may cause mayhem in file data due to different paths taken in the system software stack. Reading mapped data by `memcpy` or `read` is also likely to leak stale or corrupted data.

Tail latency. Acumen lazily triggers deployments for read requests and the deployment cost is amortized. Though, Acumen incorporates an additional flag in `ato_memcpy` for applications that are highly sensitive of a one-time longer tail read latency. Given a set flag, Acumen eagerly deploys data after each write while applications can reorganize and optimize their source codes such that they hide the latency of deployments by doing something else before launching a read request.

V. Evaluation

Setup. We have implemented and tested Acumen on a server with Intel Optane pmem in 1024GB installed. The CPU is Intel Xeon Gold 6342. The OS is Ubuntu 21.04 with kernel 5.4.206 while the compiler is GCC/G++ 10.3.0. We mount NOVA file system on pmem with a default page size of 4KB.

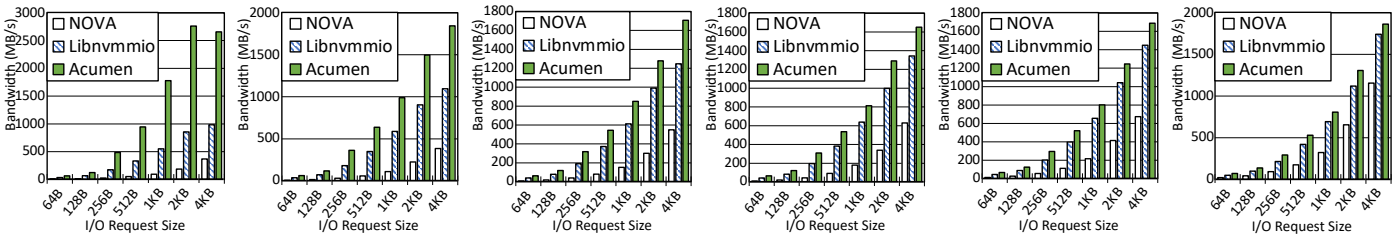
We evaluate Acumen to vanilla NOVA and Libnvmmio which are both open-source. In line with the code of Libnvmmio, all three perform pmem writes in non-temporal stores [5]. Acumen and Libnvmmio make atomic updates directly with `memcpy` variants while NOVA needs an explicit `msync` invoked after a write. To have the same level of failure atomicity, we call `msync` after `memcpy` for NOVA. We choose Fio [17] as the micro-benchmark. For macro-benchmarks, we run YCSB [18] and TPC-C [19] on SQLite 3.39.0 [16] configured with the memory-mapped I/O path. We comprehensively configure these benchmarks to issue both full- and sub-page I/O requests [6]–[8]. The main metric to measure performance is throughput.

A. Micro-benchmark

Fio has an *mmap* I/O engine that generates memory-mapped I/Os. We make it operate with a 8GB file for 600 seconds while varying the request size from being sub- to full-page and the ratio of random read and write requests. Fig. 3 captures three designs' throughputs (bandwidths in MB/s) in six diagrams.

Fig. 3a shows how three designs handle an absolute write-intensive workload that keeps issuing random write requests. Take 1KB request size for example. The throughput of Acumen is 18.9× and 3.2× that of NOVA and Libnvmmio, respectively. The superb performance of Acumen is mainly because it has

[1] https://www.man7.org/linux/man-pages/man3/ftruncate.3p.html
[2] https://www.man7.org/linux/man-pages/man2/fallocate.2.html
[3] https://www.man7.org/linux/man-pages/man7/shm_overview.7.html



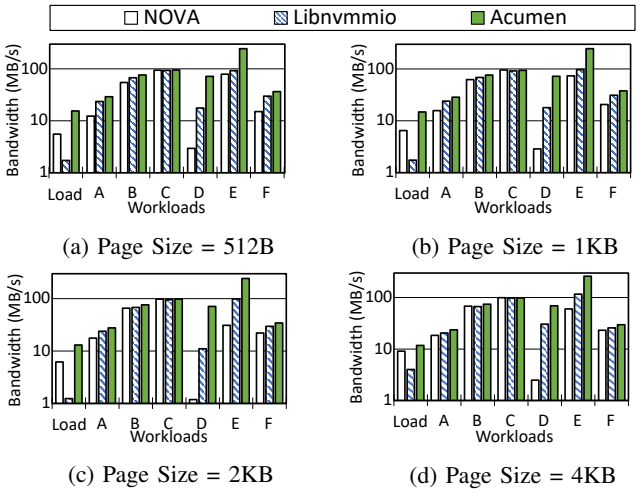
(a) Write Only (b) Read/Write = 2:8 (c) Read/Write = 4:6 (d) Read/Write = 5:5 (e) Read/Write = 6:4 (f) Read/Write = 8:2

Fig. 3: A Comparison among Acumen, Libnvmmio, and NOVA by Testing with Fio Random Read/Write Workloads

minimized pmem writes and simplistic management strategy. Given a write-only workload, Acumen does not deploy data but only writes modified data into either RefPages or original file page. Comparatively, NOVA has to copy and memory-map all file pages as replica pages before writing down newer data [6]. It demands an explicit `msync` and copies newer data back to make an atomic update. The costs of copying and `msyncs` are considerable. The reason why Acumen outperforms Libnvmmio is twofold. Libnvmmio's hybrid logging employs an undo log by default and may switch to redo logging in case that it has observed a higher write ratio ($\geq 40\%$) in the recent epoch. Undo logging writes the same volume of data to the log and and file page, respectively, on the critical path. Redo logging logs newer data and checkpoints the data to file pages. Both logging modes must write the same volume of data twice in order to achieve failure atomicity and, more important, eagerly ensure that the latest valid data stays at the mapped addresses, thereby incurring massive pmem writes. However, the lazy deployment of Acumen transiently decouples writes from reads. It writes updated data only once but delays necessary deployments until applications raise a purposeful read request. This properly matches the access pattern of random writes captured in Fig. 3a. We have recorded the volume of I/Os executed by three designs to complete the random write workload. Acumen has conducted dramatically fewer I/Os than the other two. Without loss of generality, we still illustrate with 1KB write requests. The quantity of data Acumen has written is just 20.2% and 48.6% that of NOVA and Libnvmmio, respectively. This in turn justifies the efficacy and effectiveness of Acumen.

Additionally, in Fig. 3a, Acumen yields the highest performance with 2KB request size instead of 4KB. This observation aligns with experimental results recorded by other researchers when they were performing non-temporal stores with Intel Optane pmem [12]. It also suggests to applications on how to configure and tune their request sizes in doing memory-mapped I/Os with real-world pmem products for higher performance.

As shown by Fig. 3b to Fig. 3f, with an increasing ratio of read requests, Acumen keeps yielding superior performance. For example, on serving the read-intensive workload with a ratio of 80%/20% for read and write requests, the overall throughput of Acumen is 2.5× and 1.2× that of NOVA and Libnvmmio, respectively, with the 1KB request size. The reason why the performance gap narrows is twofold. Firstly, the impact of write requests on the overall performance is substantial but decreases. Copying data to finish write requests inevitably hurts NOVA's throughput. The `msyncs` NOVA needs for failure



(a) Page Size = 512B (b) Page Size = 1KB

(c) Page Size = 2KB (d) Page Size = 4KB

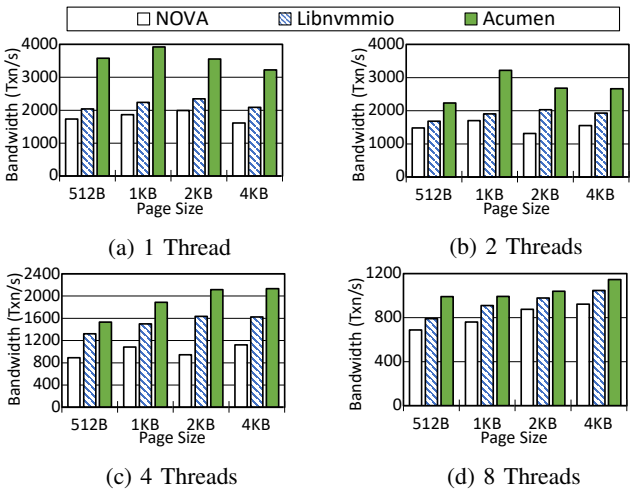
Fig. 4: A Comparison by Testing with YCSB Workloads

atomicity impose barriers that block both afterward write and read requests. Libnvmmio proceeds with undo logging for this read-intensive workload but writes the same volume of data twice on the critical path. However, writes just take up 20% of all requests. Secondly, Libnvmmio with undo logging and NOVA directly read data at mapped addresses, while a higher ratio of read requests might engage Acumen more in deploying data. Although it amortizes the deployment cost over following read requests on the same page, the likelihood of successive accesses onto one page may not be high within Fio's synthetic random read pattern. Though, Acumen still prevails over the other two designs on joint write and read performances.

B. *Macro-benchmark*

We choose SQLite as the testbed as it is widely used from data centers to smart phones. We set it with memory-mapped I/Os in a sufficient maximum `mmap` size of 120GB. The journal mode is `TRUNCATE`. SQLite writes data with the database in a unit of database page, for which we set four sizes (see Fig. 5).

YCSB. We choose YCSB for two purposes. One is to evaluate Acumen with workloads in which write and read requests are continually issued according to realistic semantics. YCSB has six typical workloads that can be found in real-world applications. For example, its workload A is known as SessionStore that records a user's recent actions in a session. The other purpose is to test the adaptability and scalability of Acumen. Take loading data for instance. The SQLite database file dynamically fluctuates from 11GB to 20GB. Each workload's runtime access behavior is also regularly changing.



(a) 1 Thread (b) 2 Threads

(c) 4 Threads (d) 8 Threads

Fig. 5: A Comparison on TPC-C Workload with Multi-threads

Fig. 4 captures the throughputs (bandwidths in MB/s) of Acumen, Libnvmmio, and NOVA when they load data and process six workloads. We run ten million records and operations with the default 1KB key-value size. Note that the Y axis Fig. 4 is in the logarithmic scale. As shown by four diagrams with varying database pages, Acumen consistently outperforms NOVA and Libnvmmio. For example, with workload A that has 50%/50% update/read running atop database pages in 512B, Acumen's throughput is 2.4× and 1.2× that of NOVA and Libnvmmio, respectively. With workload C that is read-only, three designs achieve comparable performances. The reason why Acumen yields higher performance with YCSB workloads is twofold. Firstly, it reduces write amplifications and avoids pmem writes unneeded for read operations, which is a star contrast to NOVA and Libnvmmio. Secondly, Acumen is efficiently adaptive in doing with the changes of underlying database file and each workload's access behavior. For example, when the database file size changes, all three need to remap the file. Acumen easily resizes the reflection file (see Sections IV-E). NOVA must copy back data and recopy it to new replica pages. Libnvmmio, however, is even more inefficient than NOVA due to checkpointing data upon `munmap` and building a new multi-level indexing structure in line with the resized database file. This also explains why Libnvmmio's throughput is lower than that of NOVA in Fig. 4.

TPC-C. TPC-C [19] is an OLTP workload that differs from YCSB workloads. It has five types of transactions composed of select, insert, update and delete operations. More important, we configure multi-threads when running it to further evaluate Acumen. We set eight million transactions. We vary the number of threads and also the database page size of SQLite. As shown in Fig. 5, all three's performances degrade with the increase of threads due to more lock/unlock operations enforced at multiple software layers. Whereas, Acumen still exhibits consistently higher throughput. For example, with four threads and 2KB database page, it completes 124.1% and 29.3% more transactions per second than NOVA and Libnvmmio, respectively. This confirms Acumen's efficacy in supporting concurrent memory-mapped I/Os for realistic multi-threading applications.

VI. Conclusion

In this paper, with memory-mapped files stored in pmem, we develop Acumen. Acumen pairs a reflection page to a mapped file page and makes them alternately receive updates by writing data only once to preserve the failure atomicity. It lazily deploys valid data into mapped addresses only when applications raise a read request. Extensive experiments show that Acumen substantially outperforms state-of-the-art works.

Acknowledgement

This work was jointly supported by National Key R&D Program of China No. 2022YFB4401700, Nature Science Foundation of Shanghai under Grant No. 22ZR1442000, and ShanghaiTech Startup Funding. The authors are also grateful to anonymous reviewers for their insightful comments.

References

[1] Micron, "NVDIMM: Persistent memory performance," https://media-www.micron.com/-/media/client/global/documents/products/product-flyer/nvdimm_flyer.pdf, December 2017.

[2] SK Hynix, "SK Hynix developed the world's highest density 16GB NVDIMM," https://news.skhynix.com/sk-hynix-developed-the-worlds-highest-density-16gb-nvdimm/, October 2014.

[3] Dell, "Dell EMC NVDIMM-N persistent memory: user guide," https://dl.dell.com/topicspdf/nvdimm_n_user_guide_en-us.pdf, February 2021.

[4] Hewlett Packard Enterprise, "HPE NVDIMMs," https://www.hpe.com/psnow/doc/c04939369.html, November 2021.

[5] S. R. Dulloor *et al.*, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014.

[6] J. Xu *et al.*, "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, Feb. 2016, pp. 323–338.

[7] C. Wang *et al.*, "LAWN: Boosting the performance of NVMM file system through reducing write amplification," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, p. 1–6.

[8] J. Choi *et al.*, "Libnvmmio: Reconstructing software IO path with Failure-Atomic Memory-Mapped interface," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, Jul. 2020, pp. 1–16.

[9] Z. Zhang *et al.*, "Unified DRAM and NVM hybrid buffer cache architecture for reducing journaling overhead," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 942–947.

[10] Z. Lu *et al.*, "Accelerate hardware logging for efficient crash consistency in persistent memory," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 388–393.

[11] Everspin, "Everspin releases highest density MRAM products to create fastest and most reliable non-volatile storage class memory," https://www.everspin.com/sites/default/files/Everspin%20Releases%20Highest%20Density%20MRAM%20Products%20FINAL%20041216.pdf, April 2016.

[12] J. Yang *et al.*, "An empirical guide to the behavior and use of scalable persistent memory," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, ser. FAST'20, USA, 2020, p. 169–182.

[13] S. Scargall, *Programming persistent memory: A comprehensive guide for developers*, 1st ed. Berlin, Germany: APress, 2020.

[14] D. Liu *et al.*, "Application-specific wear leveling for extending lifetime of phase change memory in embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 10, pp. 1450–1462, 2014.

[15] M. Zhao *et al.*, "SLC-enabled wear leveling for MLC PCM considering process variation," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.

[16] SQLite, "SQLite," https://www.sqlite.org/index.html.

[17] J. Axboe, "Fio - flexible I/O tester," https://github.com/axboe/fio.

[18] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154.

[19] Transaction Processing Performance Council, "TPC Benchmark C, standard specification version 5.11," https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf, 2010.