CoinPurse: A Device-Assisted File System with Dual Interfaces

Zhe Yang, Youyou Lu*, Erci Xu, Jiwu Shu
Department of Computer Science and Technology, Tsinghua University
{yang18@mails., luyouyou@, shujw@}tsinghua.edu.cn, jostep90@gmail.com

***Abstract*—Block I/O serves as a classic interface for accessing storage devices with portability. But it can also cause extra overhead by enforcing transferring data in the unit of blocks. In this paper, we present CoinPurse, a device-assisted file system with dual interfaces. By leveraging non-volatile memory (NVM) in SSD, CoinPurse manages to adaptively persist writes through both the block I/O and a byte-addressable partial update interface. In addition, we also develop a set of techniques to overcome hardware limitations and resolve possible consistency conflicts. Evaluation shows that CoinPurse outperforms F2FS, a popular flash-optimized file system, by up to 33.2%.**

I. Introduction

File system is a staple in today's storage stack for providing a significant abstraction, *file*. Yet such convenience comes with a price. The block interface prevents the host from further probing the device internals and thus limits the possibilities of leveraging the latest features. Specifically, recent breakthroughs of the Solid-State Drives (SSD), such as the on-chip Non-Volatile Memory (NVM) and the byte-addressable interface, are still beyond the reach of traditional file systems.

Unsurprisingly, reconstructing a file system to adopt the internal NVM and the byte-addressable interface is challenging. First, due to the already-crowded internal space and manufacturing cost concerns, the capacity of the on-chip NVM is usually small (e.g. a few to several hundred MB [1]). As a result, it becomes infeasible to directly utilize it for data persistence to achieve high performance. Second, to ensure backward compatibility, introducing the byte-addressable access indicates file system now can communicate with the underlying devices in two fashions (i.e. block and byte). Hence, the ordering conflict arises as independent I/Os from the two interfaces may try to modify the same piece of data simultaneously. Third, crash consistency becomes complicated. In the face of an unexpected shutdown (e.g. power outages), the in-processing data can be scattered among both the NVM and the NAND region. Recovering the file system from ongoing writes to an all-or-nothing state requires careful design.

In this paper, we present CoinPurse, a device-assisted file system with dual interfaces, targeting for high performance and consistency on SSD. CoinPurse is a log-structured file system. It additionally extends the traditional block interface, by selectively issuing a subset of write requests via the byte-addressable interface. Specifically, we design a set of practical techniques to overcome the above challenges as follows.

First, regarding limited NVM capacity, the CoinPurse filters write requests to the SSD based on thresholds. A significant proportion of partial updates use the byte-addressable interface while others are routed through the traditional block I/O. Our main insight is that file systems typically rely on synchronous writes for data durability and consistency. Previous studies show that such writes play a dominant role in the overall time consumption [2], [3]. More importantly, a great proportion of such synchronous writes are merely partial updates, where only part of a block is updated [4]. Thus, by aggregating small writes in the on-chip NVM and avoiding expensive synchronized I/O, CoinPurse can greatly boost the I/O performance.

Second, CoinPurse leverages the log-structured feature to ease the ordering conflicts by the intrinsic out-of-place updates. Yet, when two writes are issued before and after a garbage collection, it is still possible the two requests may target the same address. In this case, CoinPurse delays to invalidate a block if it is used by requests in the NVM. This omits conflicted write requests via block interface.

To guarantee crash safety, CoinPurse organizes the correlated writes requests in the format of a record in the NVM. Upon crash, the CoinPurse restores to consistency by replaying the record to avoid incomplete data flushing. Note that crash consistency led by component (e.g. NVM faults) is beyond the scope of this paper and thus not discussed.

To sum up, our major contributions are listed as follows.

- We present CoinPurse to enable fast and safe writes via a combination of two interfaces, block and byte-addressable I/O, with device assistance.
- We design and implement a set of techniques, including threshold-based filtering, lazy invalidation and transaction-style format, to provide high performance with the consistency guarantee.
- We implement CoinPurse and evaluate it against prevalent file systems. Evaluation on macro benchmarks shows that our proposed CornPurse system outperforms F2FS, a popular flash-optimized file system, by up to 33.2%.

II. Background

A. Flash File Systems

SSD builds (e.g. NAND flash memory), runs (e.g. flash translation layer) and interconnects (e.g. NVMe) differently

*Youyou Lu is the corresponding author. This work is supported by National Key Research& Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 61772300, 61832011), Huawei Innovation Project (Grant No. YBN2019125112).

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

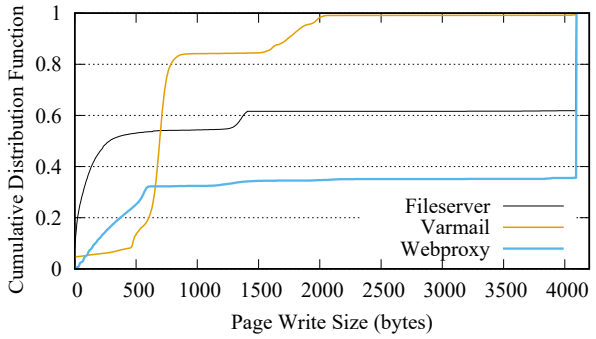


Fig. 1: The Cumulative Frequency of Page Write Sizes [4]

from the traditional spinning drives. Researchers reconsider the file system design for SSD to embrace its uniqueness.

F2FS [5] builds on the normal SSD and adapts designs atop flash characteristics. It employs the append-only log structure and a flash-friendly on-disk layout. It eliminates the update propagation issue through a node address table (NAT). OFSS [6] reconstructs the flash storage stack and exposes raw flash to software (i.e. open-channel SSD). It proposes an object flash translation layer in the host, and mitigates write amplification with co-designed mechanisms. DevFS [7] presents an aggressive design to embed the file system within the SSD. User-space applications directly access on-disk files via the shared library, bypassing the kernel.

Our proposed CoinPurse aims at adding a small extension in SSD, so as to assist the file system with partial updates.

B. Byte-Addressable Interface on SSD

The byte-addressable accesses to the SSD rely on collaboration between host and device. Memory-mapped I/O (MMIO) enables the CPU to perform memory-like accesses to peripheral devices. The device exposes its internal memory region upon interconnects and protocols. For instance, NVMe defines Controller Memory Buffer (CMB), a general-purpose region of controller memory [8]. The CMB is exposed as part of a PCIe Base Address Register (BAR), with location and size indicated in registers to allow MMIO operations.

Academia and industry further integrate non-volatile memory, e.g. capacitor-backed memory, in an SSD [1], [9], [10]. Bae *et al.* build 2B-SSD [11], a dual, byte- and block-addressable SSD, which utilizes the inner NVM and the byte-addressable interface together. FlatFlash [12] moves further to use a byte-addressable SSD as part of the main memory. The two works have to load and hold all relevant flash pages in the NVM for updating, which causes sub-optimal utilization of precious NVM resources in SSD.

C. Partial Updates in File Systems

File systems organize memory data in 4KB pages, but not all writes to pages are perfectly aligned with the page size. Figure 1 shows the cumulative distribution of metadata page write sizes of three workloads. For example, over 80% metadata writes in Varmail are smaller than 800 bytes.

On the other side, file systems manage and access the storage device in blocks (e.g. typically 4KB on SSD). Partial

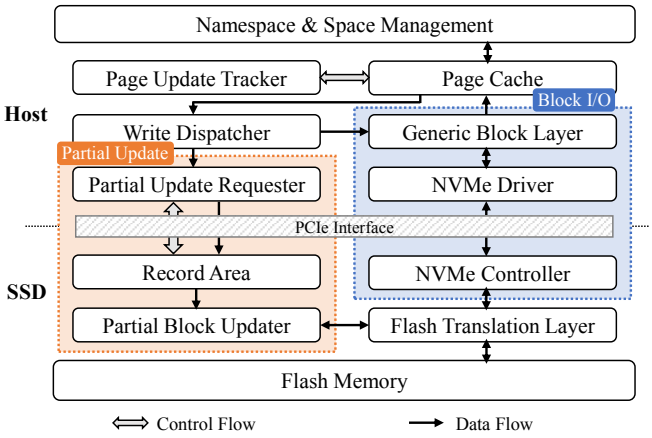


Fig. 2: The CoinPurse Architecture

writes to memory pages thus lead to partial updates to device blocks during persistence. But the block interface forces an entire block write even for a small partial update, either the original block or a new-allocated block. These writes incur severe performance overhead [2], especially when workloads trigger intensive synchronous procedures like `fsync`.

III. Design

This section illustrates the CoinPurse in detail. We start with an overview, and then describe detailed design of each component. Afterwards, we further discuss how components interact to overcome existing challenges.

A. Overview

Figure 2 shows the overall CoinPurse architecture of CoinPurse. Its components are described later in § III-B. CoinPurse consists of two main parts, a host-side file system and a device-side extended SSD. The host can use two interfaces, the classic block I/O and the partial update interface (built on atop the byte-addressable interface), to SSD.

While the block I/O path remains unchanged, writing in the partial update interface proceeds as follows. CoinPurse firstly maintains the update ranges of the page to record dirty parts, and then writes pages to allocated blocks during persistence. Based on the proportion and size of dirty data, the host issues write requests to the corresponding interface. For a request through the partial update interface, the host generates and writes it as an update record to the Record Area (NVM region exposed by the SSD). An update record contains only the updated parts of a page and a small amount of necessary metadata. Further, the SSD reads records in the Record Area, reconstructs data and writes them to destination blocks.

There are several challenges in the design of CoinPurse.

1) **Limited Size of NVM** indicates that the Record Area can get full easily with incoming requests (§ III-C).
2) **Ordering Consistency**, the writes from the two interfaces can target the same location simultaneously (§ III-D).
3) **Crash Safety**, a must-have feature for a robust file system (§ III-E).

B. CoinPurse Architecture

We first present how data are organized in CoinPurse. CoinPurse is a log-structured file system and inherits basic structures from F2FS [5]. F2FS divides and manages the device space in 4KB blocks, in line with the typical page size. It classifies data to three categories, *meta*, *node* and *data*, and stores different types of data in separate areas on the SSD. *meta* contains metadata of the file system, which are updated in place, such as the super block and the mapping table. *node* and *data* occupy most of the device space, and are updated out of place in a log-structured way. *data* blocks store data of files (including directory files), while *node* blocks contain inodes and indices of *data* blocks.

Next, we go over each component of the CoinPurse by describing their functionalities. The Page Update Tracker maintains modified ranges of each page in the Page Cache. During persistence procedures, CoinPurse generates write requests to save data of pages to allocated blocks. The Write Dispatcher dispatches writes to either the partial update interface or the block I/O. Partial Update Requester maps and manages the Record Area in the host. It also submits an update request to the SSD. The Partial Block Updater retrieves and digests records from the Record Area, to destination addresses. Specifically, we present the detailed procedures as follows.

Page Update Tracker maintains the update ranges for each page, so that CoinPurse knows update parts of each block when writing page data to allocated blocks on SSD. Although CoinPurse writes a *data* page sequentially, it may also modify scattered parts in a *node* page. For example, an inode stores attributes at the beginning and indices at the end. The Page Update Tracker thus need to record multiple ranges based on the update pattern to cover all dirty parts. The tracker initializes update ranges of a page in allocation, adjusts ranges during each update, and resets ranges after flushing data.

Write Dispatcher receives all writes requests from the Page Cache, and dispatches them to either the Partial Update Requester or the Generic Block Layer. Due to the limited size of the Record Area, the Write Dispatcher decides the suitable interface of a write request by considering the dirty size and dirty ratio (see § III-C for details).

Partial Update Requester manages the Record Area and issues partial update requests. Figure 3 shows Record Area management via MMIO. CoinPurse organizes the Record Area as a circular queue of partial update records with head and tail pointers. A record delivers sufficient information to conduct a partial update, and the record format is described in § III-C. The Partial Update Requester tries to allocate space for a write from the Write Dispatcher. Afterwards, the requester generates a update record, appends it to the queue, and finally writes the tail pointer (*Arrow W* in Figure 3) to notify the SSD. At this point, the write request is persisted and returned. On the other side, the SSD consumes records and increases the head pointer, which is read by the host (*Arrow R* in Figure 3), at the end of a persistence procedure (§ III-E).

Record Area acts as a shared persistent memory between the host and the SSD. It is based on the NVM in SSD, and exposed to the host through the PCIe BAR. The Record Area is organized as a circular queue as aforementioned. The space between the device head and tail is accessible exclusively to the device. The rest of the queue can only be touched by the host. The host and the device coordinate on the queue state through the *head* and *tail* pointers.

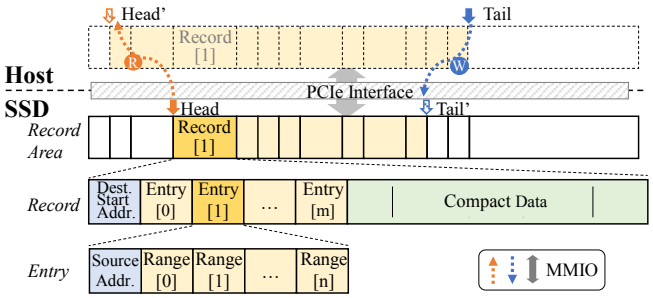


Fig. 3: Data Format on the Record Area

Partial Block Updater digests records in the Record Area asynchronously. The SSD invokes it when the Partial Updater Requester writes the *tail*. When processing a record, the updater iterates over all entries and handles them in three steps. 1) Allocate buffer for blocks, and initializes them with zero or data from source addresses. 2) Locate all update ranges and modifies them with corresponding data. 3) Write new data of blocks to destination addresses. When the processing of a record completes, the updater increases the head pointer.

C. Limited Size of the Record Area

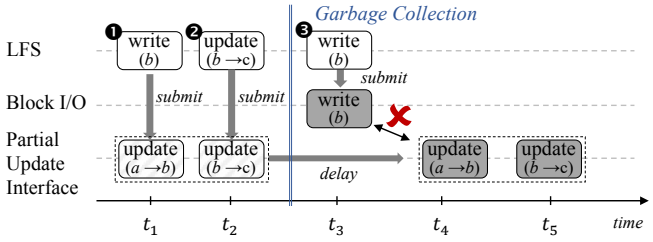
The NVM capacity in SSD is relatively small [1] and easily gets full, so CoinPurse may not push all partial-update writings to the Partial Updater Interface. To tackle the issue, the Write Dispatcher chooses critical writes based on the dispatch policy, and the Record Area stores data in a space-efficient format.

Write Dispatch Policy. A write request contains a series of pages with continuous destination block addresses. We define two metrics, dirty size S_{dirty} and dirty ratio R_{dirty}, and set two thresholds respectively for dispatch policy. The dirty size S_{dirty} of a write request is the total dirty size of all pages. The dirty ratio R_{dirty} is S_{dirty} divided by the total size of pages S_{all}. Suppose that the block I/O consumes t_{block} time per byte to submit a write request, and the partial update interface $t_{partial}$. Then we have the $Speedup$ using partial update interface against the block I/O

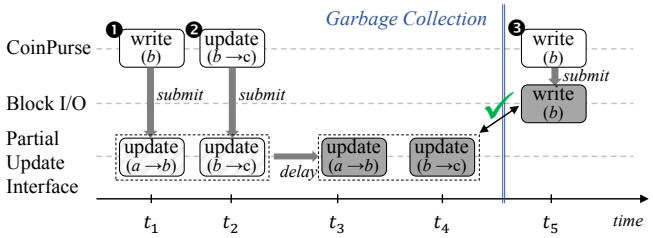
$$Speedup = \frac{S_{all} \cdot t_{block}}{S_{dirty} \cdot t_{partial}} = \frac{1}{R_{dirty}} \cdot \frac{t_{block}}{t_{partial}} \quad (1)$$

Since t_{block} and $t_{partial}$ are determined by underlying software and hardware, Equation 1 indicates that the dispatcher should choose a write request with a low dirty ratio to the partial update interface. So we set a threshold for dirty ratio to filter out high dirty-ratio writes. We also set a threshold for dirty size, considering fairness that a single request should not occupy too many resources.

If both metrics of a write are lower than thresholds, the Write Dispatcher sends it to the Partial Update Requester. The



(a) Write Reordering in LFS



(b) Solution in CoinPurse

Note that t_1–t_5 in (a) and (b) are not strictly equal nor drawn to scale.

Fig. 4: Out-of-Order Writes between Data Paths

requester tries to allocate space before submission, to avoid overflowing the Record Area. When the threshold check or space allocation fails, the write is redirected to the block I/O. CoinPurse therefore handles all write requests properly, either by block I/O or partial update interface.

Data Format of the Record Area is presented in Figure 3. Considering the limited size, we design the data format to utilize the Record Area efficiently. The Record Area is organized as a circular queue of update records with head and tail pointers. An update record corresponds to a write request of blocks with consecutive destination addresses, so only stores the start destination address. An entry represents partial updates to a block and contains metadata to conduct the update, including the source address and multiple update ranges (offsets and lengths). Instead of storing the whole 4KB block content, only dirty parts in blocks are glued and written at the compact data section.

D. Ordering Consistency

Forked data paths in CoinPurse, namely the block I/O and partial update, incur possible consistency issues. Each interface can provide write ordering of itself. More specifically, a file system imposes write ordering of block I/O by the `flush-and-wait`. CoinPurse leverages the single circular queue structure on the Record Area, to guarantee the ordering of the partial update interface. But writes between interfaces may be reordered, because the SSD handles records asynchronously. Moreover, the latest data can be in either the flash or the Record Area with the forked data paths, so CoinPurse must determine the location of the latest data.

Suppose an in-place update file system issues a partial update to block *b*, before writing the same block *b* via the block I/O. The former one may be reordered to execute later, so that the SSD stores old data falsely. CoinPurse eases this out-of-order issue with its inherent log structure. It always writes data out of place, before recycling a block by garbage collection. However, Figure 4a depicts the possible write reordering that still exists in a general log-structured file system (LFS). LFS writes (❶) and overwrites (❸) block *b* at t_1 and t_3, which implies an update from block *b* to block *c* (❷) at t_2. As long as ❶ or ❷ is issued through the partial update interface and executed after t_3, the system ends with inconsistency.

CoinPurse handles the above write reordering by *lazy invalidation*. In Figure 4b, block *b* becomes invalid after its data are updated to block *c* (❷) in a general LFS. By *lazy invalidation*, CoinPurse does not invalidate it immediately, if block *b* appears as the source or destination in the Record Area records. Consequently, a later garbage collection does not reclaim block *b*, so it will not overwritten. By reading the head pointer from the SSD, the Partial Update Requester knows completed update records and then scans them. For each update entry `update s→d`, CoinPurse invalidates the source block *s*, and invalidates the destination block *d*, if block *d* has been updated via block I/O. Henceforth, CoinPurse can reclaim and overwrite invalidated blocks, as they are used by neither interface (e.g. block *b*, ❸ at t_5).

Forked I/O paths can also lead to another consistency problem. CoinPurse writes data of blocks through both interfaces, but must determine how to read the latest data of each block. One straightforward yet costly way is to record which interface holds the latest data. If the latest data are in the Record Area, the host or the SSD reads updated parts and generates full block data. Alternatively, CoinPurse pins pages submitted to the partial update interface in the Page Cache to serve subsequent reads. Like *lazy invalidation*, CoinPurse unpins these pages when all related update records are digested.

E. Crash Safety

Crash safety is an essential feature for a storage system. CoinPurse ensures the durability of submitted persistent writes across a power failure or system crash. Furthermore, the system can recover to a consistent state after the power resumes or the system reboots.

Durability. As stated in § III-B, the Partial Updater Requester generates and writes records via MMIO. But writes are not guaranteed to reach the Record Area, because PCIe memory write transactions are *posted* without completions. At the end of a persistence procedure, e.g. `fsync`, CoinPurse reads the latest head value from the SSD. The read completion denotes that prior writes reach the Record Area [11], [13].

During a crash (e.g. power outage), all unfinished records in the Record Area, as well as head and tail pointers are flushed to the backup area on flash memory with the assistance of capacitors. Upon rebooting, the SSD restores the Record Area from the backup area, so as to provide persistence.

Recovery. The SSD restores the Record Area at the beginning of recovery, and then the Partial Block Updater digests records as normal. The Partial Update Requester writes a record and modifies the tail pointer atomically afterwards, so that CoinPurse encapsulates update operations in a record as a transaction and guarantees the atomicity.

Then we explain the *idempotence* of records. Suppose we have three partial update operations, 1) `update a → b`, 2) `update b → c` and 3) `update c → d`, where *a* to *d* are block addresses. It indicates that data are orginally in block *a*, and finally in block *d*, through a series of updates. Due to out-of-place update and lazy invalidation mechanisms described in § III-D, no update entries have the same source or destination block address simultaneously. Update relationships of blocks form singly linked lists, without a ring (e.g. no `update d→a`, `update c→a`, ..., in this case). So executing records any number of times produces consistent results. In this way, even if the power fails when the SSD processes part of a record, it is correct to execute the record from scratch again.

When the SSD completes digesting the Record Area, the host can read the latest data from the block interface. The SSD then starts to serve the host requests. Apart from recovery mechanisms of the SSD, CoinPurse keeps roll back and roll forward procedures in F2FS, so as to guarantee the file system crash consistency, on top of the Page Cache.

IV. Evaluation

A. Experimental Setup

We implement host-side components as a kernel module based on F2FS. We categorize blocks into four categories, inode, dentry, index node, and file data, by their update patterns. Through inspecting codes and tests, we employ three scattered ranges for the former two types, and one range for the rest. Update ranges information is recorded in the `private` field of `struct page`. The Partial Update Requester maintains the *tail* by the `compare-and-swap` instruction to allow concurrent multi-thread writes to the Record Area.

For lack of a programmable SSD platform, we simulate SSD components with a commercial SSD (named Data SSD) and a PU (Partial Updater) kernel module. All block I/Os are sent to the Data SSD directly. The PU module allocates a range of host memory to simulate the Record Area. Records are written to that area and a CMB region on an experimental SSD simultaneously, for real bandwidth and latency. The PU module also simulates the Partial Block Updater to consume records with estimated digesting latency, without persisting data to the Data SSD. Processing a record triggers random 4KB reads and a sequential write, so we estimate digesting latency with summed latency of these read and write requests.

In evaluation, we use a server configured with an Intel Xeon E5 CPU, 346GB DRAM and an Intel 750 SSD. It runs Ubuntu 18.04 Operating System with Linux kernel 4.18. The Intel SSD is equipped with capacitance to protect data in power failure [10]. We run `FIO` on the SSD, in order to assess the average latency of 4KB random read and sequential write of various sizes, as in Table I. The latency of write over 32KB is approximately proportional to the size. We set the digesting latency on the basis of these statistics. The simulated Record Area size is 64MB, the threshold of dirty size is 12KB and the dirty ratio threshold is 50%.

We compare CoinPurse against prevalent file systems, including F2FS, Ext4, and XFS. F2FS is a flash-optimized log-structured file system. Ext4 is a representative in-place-update file system, which is also the default choice of many Linux distributions. XFS employs an operation journal to ensure crash consistency. The three file systems run directly on the same SSD. We make and mount these file systems with default options, while mount F2FS without adaptive logging, so that it issues no in-place writes.

TABLE I: Intel 750 SSD Latency

Operation	Random Read	Sequential Write				
Granularity	4KB	4KB	8KB	16KB	32KB	64KB
Latency/us	18.20	11.76	14.12	18.55	36.33	66.50

B. Micro Benchmarks

We begin the evaluation with a set of common file system operations. We run seqread, randread, seqwrite and randwrite in `FIO`, and the handcrafted append benchmark. Each operation of these benchmarks accesses 512B data. The create and delete benchmarks randomly create and remove 100,000 files over 100 directories. Each operation in micro benchmarks is followed by `fsync`.

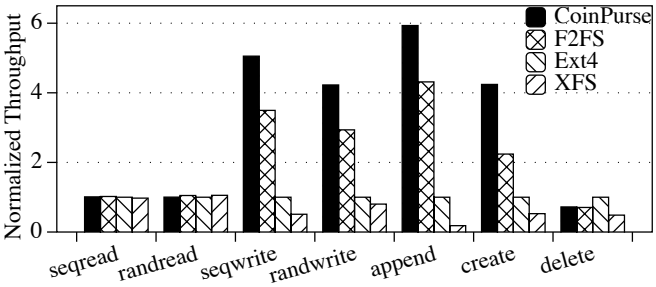


Fig. 5: Microbenchmark Throughput

Figure 5 shows the throughput normalized to Ext4. CoinPurse achieves comparable or better performance in most benchmarks. It outperforms F2FS, Ext4, and XFS up to 1.89×, 5.93× and 32.51× respectively, on seqwrite, randwrite, append, create benchmarks. These benchmarks trigger immense partial updates to both metadata and data blocks. The performance gain shows the effectiveness of dual interfaces.

However, CoinPurse and F2FS do not perform well compared to Ext4 in the delete benchmark due to a delete operation with `fsync` always triggers the costly checkpoint. XFS's performance is subpar as it logs every update to the file system without batching multiple transactions into a single log write [14]. Hence, XFS submits intensive log writes in the case of continuous small operations.

C. Macro Benchmarks

This section presents evaluation results of macro benchmarks. We choose Varmail, OLTP-insert, DBbench and Mobibench for evaluation. Varmail in Filebench [15] is a metadata-intensive workload to simulate a mail server. OLTP-insert is a workload in Sysbench [16]. It adopts MySQL, a famous relational DBMS, as the back end. DBbench runs on RocksDB, a popular key-value store system. We use the *fillsync* attached with DBbench. Mobile applications rely on SQLite database heavily to store data [17], so we also evaluate CoinPurse with Mobibench [18], which is based on SQLite.

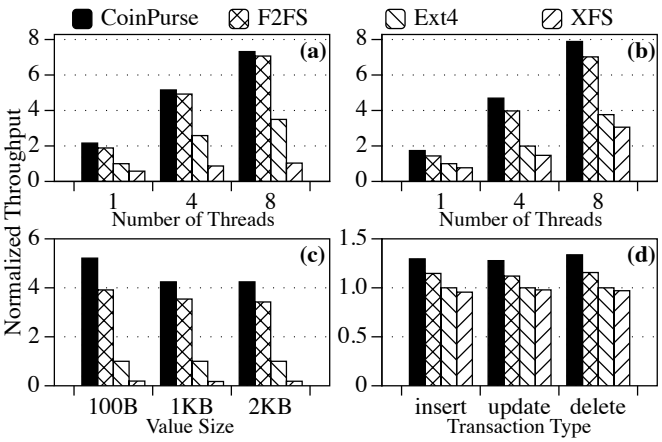


Fig. 6: Macrobenchmark Throughput

Throughput. Figure 6 a–d show throughput of Varmail, OLTP-insert, DBbench and Mobibench repectively. Results are normalized to Ext4, and 1 thread in Figure 6a–b specifically.

Varmail does a series of operations to files, including create, delete, append and `fsync`. Figure 6a shows that file systems perform discretely on Varmail. They scale well as the number of threads increases except XFS, due to intensive metadata operations. CoinPurse outperforms F2FS, Ext4 and XFS by up to 1.16×, 2.18× and 7.23× respectively.

Figure 6b presents results on OLTP-insert. Ext4 and XFS achieve similar performance in all cases. The throughput of CoinPurse is 1.18× F2FS, 2.09× Ext4 and 2.71× XFS averagely. MySQL writes redo log files in multiple 512B sectors, where consecutive writes may overwrite the same 4KB unit in a file. CoinPurse leverages the Partial Update Interface to absorb small updates, so as to bring improvement. MySQL fills files before writes, which mitigates space allocation overhead of in-place-update Ext4 and XFS. But overwriting the same 4KB block frequently incurs severe overhead on SSD.

Figure 6c compares the normalized performance of DBbench on RocksDB. The key is 16B, and the value size ranges from 100B (default) to 2KB. CoinPurse surpasses F2FS by 28.1%–33.2%, and achieves up to 5.21× throughput of Ext4, 27.1× XFS. Throughput values differ widely between file systems, because RocksDB creates and appends new log files continuously. Therefore, performance of file systems on DBbench resembles that on the append benchmark in § IV-B.

Figure 6d presents results on Mobibench. We use the WAL journal mode and the FULL synchronous mode to test 100,000 insert, update and delete transactions. Ext4 and XFS perform closely, but CoinPurse is around 30% faster than them, and 17.2% better than F2FS in average. SQLite accesses storage including journaling at the granularity of 4KB by default. Consequently, gaps between F2FS, Ext4 and XFS are not dramatic as former benchmarks.

Write Statistics. We collect statistics of *node* and *data* writes in the Write Dispatcher of CoinPurse, and present them in Table II. We select Varmail and OLTP-insert in 1 thread, DBbench with 1KB value size, and Mobibench's update workload. The left three columns cover all writes the Write Dispatcher receives. The right three columns indicate statistics of writes dispatched to the Partial Update Interface.

TABLE II: Write Statistics of Macro Benchmarks

Benchmark	All Writes			Writes to Partial Update Interface		
	Size/MB	Dirty Ratio	Avg. Write Size/KB	Size/MB	Size Proportion	Dirty Ratio
Varmail	14,647	54.22%	7.51	4,993	34.09%	9.34%
OLTP-insert	6,126	23.18%	4.81	4,905	80.07%	9.56%
DBbench	12,522	8.75%	6.40	6,449	51.50%	3.43%
Mobibench	3,515	61.31%	17.87	1,365	38.83%	25.93%

We have several observations from Table II. First, the dirty ratio of all writes is low, which is only 8.75% in DBbench. The average size per write ranges from 4.81KB to 17.87KB. These demonstrate that file systems issue immense small partial updates through block I/O. Second, the dirty ratio of writes to the Partial Update Interface is lower than that of all writes. Because the threshold in the Write Dispatcher filters out writes with high dirty ratio. Third, the Partial Update Interface handles at least 34.09% writes and reduces block I/O traffic significantly. Note that CoinPurse outperforms F2FS by 21.08% on OLTP-insert, although it dispatches 80.07% data to the Partial Update Interface. This implies that logic apart from synchronous writes in OLTP-insert causes more overhead, compared to other benchmarks.

V. Conclusion

In this paper, we present CoinPurse, a device-assisted file system with dual interfaces, the conventional block I/O and the proposed partial update interface on top of NVM on SSD. CoinPurse dispatches writes to dual interfaces dynamically, without loss of durability and consistency. Evaluation shows that CoinPurse outperforms F2FS, Ext4 and XFS in macro benchmarks, by up to 1.33×, 5.21× and 27.1× respectively.

References

[1] W.-H. K. et al., "Durable write cache in flash memory ssd for relational and nosql databases," in *SIGMOD*. ACM, 2014.
[2] D. Park *et al.*, "iJournaling: Fine-grained journaling for improving the latency of fsync system call," in *USENIX ATC*, 2017.
[3] G. Lee *et al.*, "Asynchronous i/o stack: a low-latency kernel i/o stack for ultra-low latency ssds," in *USENIX ATC*, 2019.
[4] Y. Lu *et al.*, "Supporting system consistency with differential transactions in flash-based ssds," *IEEE Trans. Comput.*, Feb. 2016.
[5] C. Lee *et al.*, "F2FS: A new file system for flash storage," in *FAST*. USENIX Association, 2015.
[6] Y. Lu *et al.*, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *FAST*, 2013.
[7] S. Kannan *et al.*, "Designing a true direct-access file system with devfs," in *FAST*. USENIX Association, 2018.
[8] NVMe specifications. https://nvmexpress.org/resources/specifications/
[9] Y. Jin *et al.*, "Improving ssd lifetime with byte-addressable metadata," in *MEMSYS*. ACM, 2017.
[10] Intel ssd 750 series datasheet. https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-750-spec.pdf
[11] D.-H. Bae *et al.*, "2B-SSD: the case for dual, byte-and block-addressable solid-state drives," in *ISCA*. IEEE, 2018.
[12] A. Abulila *et al.*, "FlatFlash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy," in *ASPLOS*. ACM, 2019.
[13] Pcie base specification revision 3.0. https://pcisig.com/specifications
[14] C. Hellwig. XFS: the big storage file system for linux. https://www.usenix.org/system/files/login/articles/140-hellwig.pdf
[15] A. Wilson, "The new and improved filebench," in *FAST*, 2008.
[16] Sysbench. https://github.com/akopytov/sysbench
[17] M. Son *et al.*, "A small non-volatile write buffer to reduce storage writes in smartphones," in *DATE*. EDA Consortium, 2015.
[18] Mobibench. https://github.com/ESOS-Lab/Mobibench