TxISC: Transactional File Processing in Computational SSDs

Penghao Sun[1], Shengan Zheng[2]✉, Kaijiang Deng[1], Guifeng Wang[1], Jin Pu[1],
Jie Yang[3], Maojun Yuan[3], Feng Zhu[3], Shu Li[3], Linpeng Huang[1]✉
[1]*Shanghai Jiao Tong University*
[2]*MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University*
[3]*Alibaba Group*
✉{shengan, lphuang}@sjtu.edu.cn

***Abstract*—Computational SSDs implement the in-storage computing (ISC) paradigm and benefit applications by taking over I/O-intensive tasks from the host. Existing works have proposed various frameworks aiming at easy access to ISC functionalities, and among them generic frameworks with file-based abstractions offer better usability. However, since intermediate output by ISC tasks may leave files in a dirty state, concurrent access to and the integrity of file data should be properly managed, which has not been fully addressed. In this paper, we present TxISC, a generic ISC framework that coordinates the host kernel and device firmware to offer a versatile file-based programming model. Under the hood, TxISC turns each invocation of an ISC task into a transaction with full ACID guarantee, fully covering concurrency control and data protection. TxISC implements transactions at low cost by leveraging the out-of-place write characteristic of NAND flash. Evaluation on full-stack hardware shows that transactions incur almost no runtime performance penalty compared with existing ISC architectures. Application case studies demonstrate that the programming model of TxISC can be used to offload complex logic and deliver significant speedup over host-only solutions.**

***Index Terms*—in-storage computing, computational SSD, transaction, programming model**

I. Introduction

In-storage computing (ISC) takes advantage of computing resources within a storage device to perform computation tasks in the host's stead. By offloading data processing to where data reside, ISC significantly shortens the I/O path, benefiting data-intensive applications. As of today, NAND flash-based solid state drives (SSDs) are the preferred vehicle of the ISC paradigm [1]. Indeed, ultra fast NAND flash is one of the major drivers behind ISC, since faster storage media amplifies the impact of software stack overhead and bus transfer latency on end-to-end performance [2]. Additionally, to relieve the host of the "erase before write" idiosyncracy of NAND flash, modern SSDs are already packing reasonable computing power in order to run the flash translation layer (FTL), making them ideal candidates to take over simple data processing jobs from host applications [3], [4].

Accessible use of ISC-enabled SSDs, or computational SSDs, calls for the development of easy-to-use ISC frameworks. Prior works explored application-specific architectures [5]–[8] and generic programming frameworks [9]–[13]. Although the former boasts superior performance in their respective domains, wide adoption is limited due to lack of generality [14]. Generic frameworks, on the other hand, allow users to offload arbitrary ISC tasks by providing general-purpose programming models. Among them, file-based programming models [11]–[13] provide better usability than block-based ones [9], [10] since most existing applications also interact with the SSD through file system abstractions, instead of directly accessing raw block devices.

However, allowing ISC tasks to manipulate files inside the SSD introduces another party that can alter data at runtime. It is thus crucial to provide system-level support of concurrency control and data protection since intermediate output by ISC tasks may leave files in a dirty state. In this regard, it is desirable to have *ACID* semantics for ISC tasks similar to database transactions. Although prior solutions have provided preliminary support of isolation among ISC tasks by means of, for example, separating the control plane and data plane [11], [12], ISC tasks interrupted by system crashes can still cause file data corruption. While it is also possible to leave the responsibility of ACID to applications by using techniques such as logging or copy-on-write, the excessive data copies could offset the benefit of adopting ISC in the first place [15]. Our goal in this paper, therefore, is to design a generic ISC framework with built-in support of file transactions.

We present TxISC, a generic ISC framework that coordinates the host kernel and device firmware to offer a versatile file-based programming model. User-defined ISC tasks can access file data inside the SSD through POSIX-like APIs while remaining agnostic to the file system in use. Under the hood, TxISC turns each invocation of an ISC task into a transaction with full ACID guarantee. TxISC creates exclusive snapshots for files that an ISC task accesses. Changes made to the file snapshots remain private until the task commits. By leveraging the out-of-place write property of NAND flash, TxISC ensures uncommitted changes can be rolled back on aborting or recovery. When a task commits, data changes are applied by updating the L2P table atomically to prevent intermediate changes from becoming visible. For durability, TxISC persists a minimal amount of mapping information that can be used to rebuild the L2P table that correctly reflects committed changes upon recovery.

We implement TxISC on full-stack hardware evaluation platform. Microbenchmark and application case studies demonstrate that TxISC can provide transaction support with almost no runtime performance penalty compared with existing ISC architectures. Furthermore, offloading computation to TxISC provides 14%-74% improvement over host-only solutions in various I/O-intensive workloads. To the best of our knowledge, TxISC is the first ISC framework that features full-fledged ACID support for offloaded tasks.

This paper makes the following contributions:

- We present TxISC, a versatile ISC framework that coordinates the host kernel and device firmware to allow offloaded computation tasks to access data with easy-to-use file abstractions.
- TxISC turns each invocation of a user-defined ISC task into a transaction with full ACID guarantee. Intermediate changes to file data are made to exclusive file snapshots and remain private until committing. Should the task fail, uncommitted changes can be rolled back. For committed tasks, TxISC ensures that data changes are applied atomically and durably.
- We implement TxISC on real hardware and evaluate it with microbenchmark and application case studies, demonstrating low runtime cost of transaction support and significant speedup over baseline systems.

II. Background and Related Work

A. SSD Basics

Solid state drives (SSDs) are built on arrays of NAND flash chips. Since programmed (written) flash pages must be erased before they can be programmed again, SSDs pack an embedded controller to run the flash transaction layer (FTL) [16]. When serving block write requests from the host, the FTL always allocates new flash pages and maps logical block addresses (LBAs) to new physical page addresses (PPAs) with the L2P table, thus avoiding frequent erase operations. When free flash pages deplete, garbage collection is triggered, during which the FTL identifies valid pages using the reverse P2L mapping stored alongside user data in the OOB area.

B. Computational SSDs

With the I/O latency of SSDs approaching sub-10μs domain, software stack overhead becomes amplified [2]. In-storage computing offers a potential solution. Fortunately, SSDs are a natural fit with the ISC paradigm, since the device controller already comes with moderate computing power [3] due to increasingly heavy firmware tasks (wear levelling, data retention handling [17], etc.). FPGAs and other domain-specific accelerators [11] have also found their way into ISC-enabled SSDs, or computational SSDs.

Prior attempts toward computational SSD system can be classified into domain/application-specific architectures [5]–[8] and generic frameworks [9]–[13]. Although the former can generally deliver superior performance by incorporating highly customized software and hardware optimizations for the targeted applications (e.g., information retrieval [5], neural network training [6], [7], data analysis in HTAP [8]), they lack flexibility in terms of use cases and may require complete revamp as the underlying application evolves.

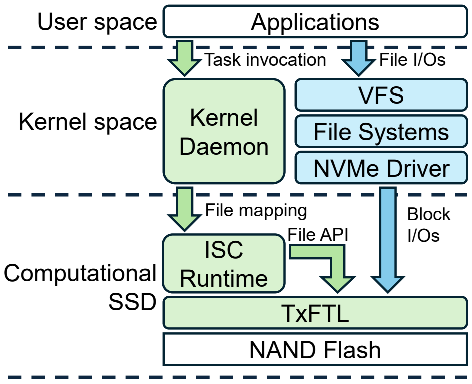

Fig. 1. TxISC architecture.

On the other hand, generic computational SSD frameworks [9]–[13] allow users to better take advantage of the programmability of computational resources within the storage device, albeit to varying extents. Among them, Willow [9] and Summarizer [10] adopt the raw block interface, where the user specifies input/output data with their LBAs. Such interface aligns well with existing NVMe infrastructure, but is arguably hard to use by application developers. Direct exposure of the block address space is also prone to data security issues. Insider [11], Metal FS [12] and λ-IO [13] build on file system abstractions, providing a more user-friendly interface.

Nevertheless, one critical functionality yet to be offered by existing solutions is transaction support for the offloaded ISC tasks. Such support is important since allowing ISC tasks to mutate data in-storage brings the risk of data corruption when multiple tasks and host applications share the same dataset, or when on-going tasks are interrupted by system crashes. It is thus ideal to have ACID semantics support for ISC tasks similar to database transactions. Therefore, our goal in this work is to build a generic computational SSD framework, offering transactional file processing capability.

III. Design

We present TxISC, a computational SSD framework that allows users to offload data processing to the storage device. TxISC coordinates the device firmware and the host kernel and provides a file-based transactional programming model. As shown in Figure 1, its main components include:

1) The device-side *ISC runtime*, providing offloaded ISC tasks with basic programming support such as memory allocation and access to on-device file data through POSIX-like APIs (both synchronous and asynchronous APIs are supported). An ISC task is simply a C function that can be loaded into the device and invoked by host applications on demand.

2) The device-side *TxFTL*, extending the conventional FTL in SSDs to handle file I/O requests from ISC tasks. By carefully managing the translation from file offsets or LBAs to target PPAs, TxFTL enforces atomicity of data updates, consistency of the file system, isolation among ISC tasks and host applications, and durability of committed changes.

```c
// host-side application
int file_checksum_host() {
  int id[2], status, checksum = -1;
  // declare ISC context
  ISCCtx c;
  // register input and output files
  id[0] = isc_register_file(&c, "/in/path", "r");
  id[1] = isc_register_file(&c, "/out/path", "w");
  // add other arbitrary arguments
  isc_put_args(&c, id, 2 * sizeof(int));
  // call ISC task
  status = isc_run_task(&c, CHECKSUM_TASK_HANDLE);
  // return checksum on success and -1 otherwise
  if (status == SUCCESS)
    isc_get_ret(&c, &checksum, sizeof(int));
  return checksum;
}
// device-side task implementation
void file_checksum_device(ISCCtx *c) {
  int id[2], size, checksum;
  // retrieve arguments and allocate buffer memory
  isc_get_args(c, id, 2 * sizeof(int));
  size = isc_file_size(id[0]);
  void *buf = isc_malloc(size);
  // read input file data and calculate checksum
  isc_file_read(id[0], buf, size);
  checksum = calculate_checksum(buf, size);
  // write checksum to output file
  isc_file_write(id[1], &checksum, sizeof(int));
  // return calculated checksum
  isc_put_ret(c, &checksum, sizeof(int));
}
```

Listing 1. An example: computing file checksum.

3) The host-side *kernel daemon*, exposing TxISC functionalities to host applications. The kernel daemon also interfaces with the VFS layer to retrieve file mappings when necessary.

Below in this section, we detail the user programming model of TxISC and how the components work together to ensure transactional file access at low cost.

A. Programming Model

TxISC provides a transactional programming model, where each invocation of an ISC task is treated as a transaction. Specifally, TxISC offers the following ACID semantics:

- *Atomicity (Section III-C)*: Changes to files are committed atomically. Ongoing ISC tasks and host applications will not see partially updated data from each other.
- *Consistency (Section III-D)*: On the host side, the kernel daemon interacts with the VFS layer only. Inside the device, only data blocks can be accessed by ISC tasks, and file system metadata remains intact. TxISC is thus agnostic to the specific file system in use (ext4, f2fs, etc.), and can inherit the file system's consistency guarantee.
- *Isolation (Section III-B)*: ISC tasks and host applications can make changes to a file concurrently, as long as changed data do not overlap. Furthermore, each ISC task operates on exclusive file snapshots created on invocation. Before the task commits, its changes to file data remain private. Should there be conflicting changes (i.e., when modified ranges overlap), the task that arrives later can optionally retry or abort and have all data changes rolled back.
- *Durability (Section III-D)*: Once a task commits, its changes to file data are guaranteed to have been persisted and can survive system failures.

We demonstrate the use of TxISC with a simple in-storage file checksum calculation example in Listing 1. Host-side application code (`file_checksum_host`) can register input/ouput files and add other arguments to an ISC context, which represents an instance of an ISC task invocation. Later, the host calls `isc_run_task` to trigger the ISC task. Once it returns, the task is guaranteed to have completed. The user can then check whether the task was successfully committed and take actions accordingly. Device-side task code (`file_checksum_device`) runs inside the computational SSD. The task can read/write files using POSIX-like interfaces, allocate scratchpad memory from the on-board RAM, perform computation in-storage, and pass return values to the host. When the function returns, the task completes and changes are committed according to the ACID semantics described above.

B. Isolating File Access

To enforce isolated file I/Os among ISC tasks and host applications, TxISC creates a snapshot for each file that is to be accessed by an ISC task upon its invocation. The snapshot remains exclusive to the task before the task commits.

Based on the observation that writing flash pages does not overwrite existing data, TxISC extends the address mapping facilities in the FTL to maintain file snapshots at low cost. As shown in Figure 2, per-file snapshot info is stored in a snapshot descriptor, which is associated with one and only one file. Multiple tasks can process a file simultaneously by sharing a snapshot descriptor. A snapshot descriptor holds two mapping structures: (1) An array of file extents queried from the VFS by the kernel daemon when the user registers a file to the ISC context. The extents map pages that contain untouched file data when an ISC task starts. (2) Delta mapping entries used to track file data changed by ISC tasks or the host. The entries are organized in a red-black tree, indexed by file offset. With such information, TxFTL can isolate file accesses from the host and ISC tasks, as described below.

Handling in-storage file I/Os. To keep partially committed data invisible to the host and other ISC tasks, in-storage file writes do not directly update the L2P table. Instead, pages allocated for the newly written data are tracked using the delta mapping entries. Besides the flash page address and the corresponding file range, a delta mapping entry also records the ID of the task from which the data originated and a timestamp (`ts`). For illustration, in Figure 2, the ISC task with ID 20 is to write data block 2 of file `foo` (TxISC supports file I/O with arbitrary size and offset; here we assume block-aligned I/O for ease of illustration). TxFTL first searches the delta mapping entries in the file's snapshot descriptor for overlapping ranges. If one such entry is found, there is potential data conflict. TxFTL then checks whether the delta

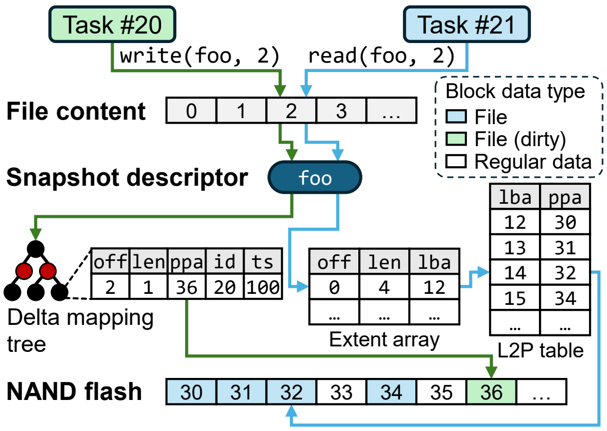

Fig. 2. Isolating file access through snapshot descriptor.

mapping is from another task by comparing the task ID. Should the IDs differ, the task that arrives later aborts or retries due to conflicting file writes. Otherwise, TxFTL allocates flash pages to house the new data and inserts a new delta mapping entry or modify existing entries accordingly.

Concurrent ISC tasks can avoid reading dirty data with the help of delta mapping. In Figure 2, another task with ID 21 tries to read block 2 of file `foo` that has been written by task 20. TxFTL searches the delta mapping tree for entries that cover data block 2. Although one such entry is found in Figure 2, it originated from task 20 and is thus omitted. TxFTL then retrieves the address of the flash page that contains untouched file data from the extent array and the L2P table, thus correctly reading clean file data.

Handling host file I/Os. With TxISC, host applications retain the ability to access files via the normal I/O path, which eventually takes the form of block I/Os to the computational SSD. Internally, TxFTL takes extra care to prevent data races from the host. When the user registers a file, TxFTL sets a flag bit in each L2P table entry covered by the file extents (not shown in Figure 2), indicating that the LBA belongs to a file snapshot. TxFTL can then identify host writes to LBAs that are part of a file snapshot by checking the bit in the L2P entry. If the bit is set, the L2P entries are not updated directly. Instead, a delta mapping entry with a special task ID representing the host is inserted to the corresponding snapshot descriptor. However, since file systems do not always write data in place (e.g., log-structured file systems), only intercepting host writes inside the device is not enough. Therefore, on the host side, the kernel daemon keeps track of file ranges written by host programs for a registered file by instrumenting relevant syscalls and maintaining an extent tree-like structure for the file in the VFS layer. This information is later used for checking conflicting writes when the task is committed. For reads, it suffices to simply identify I/O requests that target LBAs in a snapshot and retrieve the latest flash pages from the delta mappings generated by the host.

C. Atomic Task Committing

TxISC ensures transaction atomicity by making file data changes from an ISC task visible to other parties atomically when a task commits. Specifically, TxFTL tracks the status of ongoing tasks with a *global task table*. Each task table entry contains a timestamp representing the time at which the task is committed. When a task completes (its C routine returns), it is staged for committing. TxFTL first requests up-to-date file extents (in case of log-structured file systems) and file ranges written by the host from the kernel daemon to check for conflicting writes. If no conflicts are found, the commit timestamp in the corresponding task table entry is set to the current time, marking a successful commit. Afterwards, TxFTL applies the delta mapping entries to the L2P table and persists P2L mappings to dedicated flash pages for garbage collection needs. TxFTL uses locks to ensure atomicity of the propagated changes. Since this is a pure DRAM operation, the locking period is short. Once the L2P table have been updated, the task's changes to file data become globally visible.

D. Crash Recovery

For durability and crash consistency, the global task table and delta mapping entries need to survive power losses. TxISC achieves this by putting them in a DRAM region backed by capacitors so that they can be flushed to flash in the event of a power loss. Note that modern SSDs are already widely using capacitors for data reliability [18]. During recovery, TxISC rebuilds the L2P table by scanning on-disk P2L mappings and the delta mappings. When scanning the delta mappings, those with the special task ID representing the host are always applied. By doing so, TxISC inherits the file system's consistency guarantees. Those with normal task IDs are applied if and only if the recorded timestamp is earlier than the commit timestamp in the task table because an entry with a timestamp later than the commit timestamp points to pages containing uncommitted data. After this process, the L2P table is restored to a consistent state, with all writes by committed tasks applied and all writes by uncommitted tasks rolled back.

IV. Implementation

We build a TxISC prototype on full-stack hardware (refer to Table I for full configurations). We implement the kernel daemon in Linux 5.10.186. For computational SSD, we use the Daisy+ OpenSSD platform [19]. Since the OpenSSD does not come with real NAND flash, the storage space is backed by two 32GB DDR4 DIMMs. To more realistically reflect the performance of NAND flash-based SSDs, we port the page-level FTL with flash emulation capability from the FEMU emulator [20] and extend it to integrate TxFTL functionalities. The OpenSSD has a quad-core ARM Cortex-A53 controller (as part of a Zynq Ultrascale+ SoC). In our current implementation, flash latency emulation occupies a controller core, and NVMe request handling and TxFTL use another dedicated core, leaving 2 cores for ISC tasks. Our implementation of TxISC includes 331 lines and 9754 lines of host-side and device-side code, respectively.

TABLE I
EVALUATION SYSTEM CONFIGURATIONS.

Platform	Item	Configuration
Host	CPU	2× Intel Xeon Gold 6240
	Memory	384GB DDR4
	OS	Ubuntu 22.04 LTS (Linux 5.10.186)
Device	CPU	4× Cortex-A53 @ 1.5GHz
	Memory	2GB LPDDR4
	Host interface	NVMe over PCIe 3.0 ×8
	Storage backend	2× 32GB DDR4 DIMM
	Flash layout	16KB page, 512 pages per block, 547 blocks per die, 2 dies per channel, 8 channels
	Flash latency	80μs read, 400μs write, 4ms erase

V. EVALUATION

In this section, we present the evaluation results of our TxISC prototype, focusing on the performance cost of our FTL-level extensions, and the expressiveness of the proposed programming model for offloading I/O-intensive tasks in real-world applications.

A. Setup and Methodology

Full configurations of our evaluation platform are listed in Table I. As discussed in Section IV, NAND flash behavior is emulated using a page-level FTL [20]. We configure the flash layout to emulate 7% over provisioning. To evaluate the overhead of transaction support and file request handling, we compare TxISC with two baseline ISC architectures:

- *FileISC* uses the same file-based programming model as TxISC but removes transaction support. Changes to file data from ISC tasks are committed directly to the L2P table. Insider [11] and λ-IO [13] adopt similar architectures in their device-side design.
- *BlockISC* further removes file abstractions from FileISC. ISC tasks can access flash data with raw block addresses. Willow [9] and Summarizer [10] follow this design.

We also include the host-only solution where applicable in the experiments.

We evaluate TxISC and the baselines with microbenchmarks and application case studies. In microbenchmarks, we characterize the overhead of transaction support by testing in-storage I/O performance. For real-world applications, we follow the programming model of TxISC and offload data-intensive tasks to the computational SSD device to measure performance gains.

B. Transaction Overhead

Figure 3 shows the aggregate bandwidth of in-storage random file I/Os with different read/write ratios and queue depths. I/O size is set to 16KB to align with the underlying flash page size. We use random I/Os with small request size to evaluate transaction overhead because such workloads put the most pressure on TxFTL.

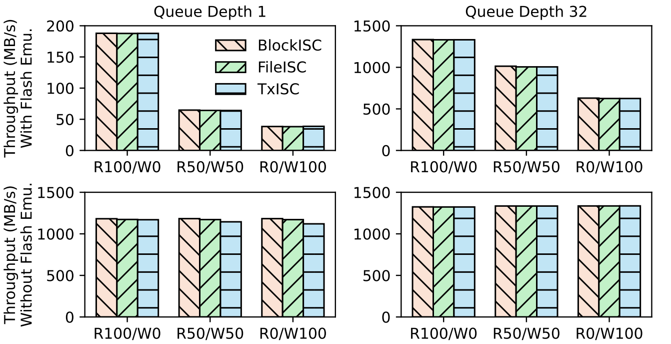

Fig. 3. In-storage 16KB random file I/O performance.

In TxISC, transaction support only affects I/O performance when the workload has write requests. However, with flash emulation enabled, the performance degradation is negligible compared with FileISC and BlockISC (within 0.3%), even in 100%-write workloads. This confirms that maintaining file snapshots and delta mappings does not incur significant performance penalty when serving in-storage I/Os. In terms of on-board memory cost, each delta mapping entry has a fixed size of 22B in the current implementation (1B task ID, 4B offset, 1B length interpreted as the number of continuous flash pages, 4B PPA, 4B timestamp, 8B tree node) and accounts for the vast majority of runtime memory consumption. Each entry can map up to 128 flash pages. With 16KB flash pages, the memory overhead is 0.001% to 0.134% of written data.

To fully stress the software stack, we disable flash emulation by establishing a direct mapping between LBAs and the DDR4 backend address space and removing all latency simulations. This setup pushes pressure on the device firmware to the extreme. In the 100%-write workload with queue depth set to 1, we see a 4.3% and 5.2% performance drop compared with FileISC and BlockISC, respectively. This is because this workload represents the worst-case scenario for TxISC since each request will generate a new delta mapping entry, quickly populating the delta mapping tree. Limiting the I/O queue depth to 1 amplifies the impact as the extra bookkeeping work will be entirely on the critical path of the I/O request. In line with our analysis, the performance gap between FileISC and BlockISC drops to 2.4% and 3.2% when write proportion is reduced to 50%. With queue depth set to 32, the performance penalty disappears since the bottleneck shifts to the hardware as I/O queues become deeper. Since most real-world workloads have mixed read/write profiles and exploit parallel I/Os, we conclude that TxISC can provide transaction support at low runtime cost.

C. Application Case Studies

We evaluate the performance of TxISC using three applications with distinct I/O and computation profiles: bitmap decompression, graph BFS and LSM tree compaction. The results are reported in Figure 4. We also break down the applications' execution time into I/O and computation in Figure 5. We do not include BlockISC here because all applications use files as input and output.

Bitmap decompression. This application reads a compressed bitmap file and decompresses it into another file. We use the same file format as in [11] and [13]. In TxISC, the entire decompression process is offloaded to the computational

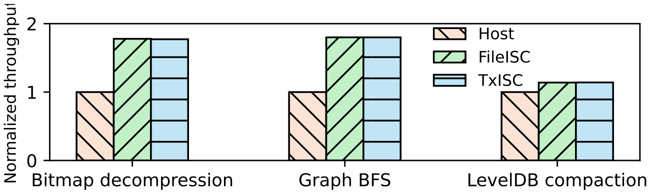

Fig. 4. Application throughput.

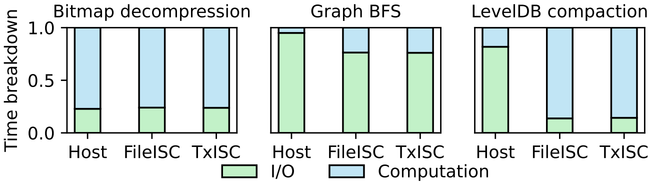

Fig. 5. Application execution time breakdown.

SSD. Here we use a synthetically generated bitmap file (16MB in size) that decompresses into 256MB. This application features strictly sequential read/write from/to a single input/output file with little computation required.

By processing data in-storage, TxISC enjoys faster access to the storage media than the host, thus outperforming the host-only solution by 1.74×. Compared with FileISC, TxISC is able to maintain 99.6% of the throughput, again demonstrating the low performance cost of transaction mechanisms. Since there is no data dependency between consecutive outputs, all tested systems can leverage buffered and asynchronous I/O, pushing I/O overhead down to 22%-24%.

Graph BFS. This application performs a breadth-first search on a graph. The graph is stored in a file using the CSR format. We use the LiveJournal social network from SNAP [21] as the dataset. This application is a typical pointer-chasing workload with small, random I/Os and moderate computation.

In the host-only solution, I/O time takes up to 95% of the program execution time due to random I/O pattern and data dependency. However, we do not see higher performance gains from offloading compared to bitmap decompression. This is because non-I/O part of the application becomes slower inside the SSD, such as maintaining the set of visited vertices and updating the neighbor list. Since there is no write involved, TxISC exhibits the same performance as FileISC.

LevelDB compaction. This application offloads SSTable compaction in LevelDB. We implement SSTable file parsing and formatting inside the SSD and expose a compaction ISC task to the host. The task takes the SSTable files to be compacted as input, performs compaction in-storage, and returns the list of newly generated SSTable files. We use LevelDB's built-in db_bench utility and the default test suite. This application needs to read and write multiple files and perform a considerable amount of computation along the way.

TxISC can finish a compaction operation 14% faster than the host. Although 81% of compaction time is spent on I/O when handled by the host, the wimpy CPU in the SSD shifts bottleneck to computation, where I/O only takes 13%. The final performance improvement is thus not as significant. Nevertheless, faster compaction contributes to a 27% improvement in the 99th percentile tail latency of random insertions. We do not observe any performance degradation in TxISC compared with FileISC since the extra overhead in transaction handling is masked by the large amount of computation.

Acknowledgment

This work is supported by National Natural Science Foundation of China (NSFC) (Grant No. 62332012, 62227809, 62302290), the Fundamental Research Funds for the Central Universities, Shanghai Municipal Science and Technology Major Project (Grant No. 2021SHZDZX0102), and Natural Science Foundation of Shanghai (Grant No. 22ZR1435400), and Alibaba Group through the Alibaba Innovative Research (AIR) program.

VI. Conclusion

In this paper, we present TxISC, a generic ISC framework that provides transactional file processing capability and features full ACID support, thus offering concurrency management and data protection as the storage device gains the ability to mutate data. Transaction support is implemented at low cost by leveraging the out-of-place write characteristic of NAND flash. Full-stack hardware-based evaluation demonstrates that the performance penalty of transactions is negligible, and the programming model is expressive enough to offload I/O-intensive tasks with complex logic in real-world applications.

References

[1] J. Do, S. Sengupta, and S. Swanson, "Programmable solid-state storage in future cloud datacenters," *Communications of the ACM*, vol. 62, no. 6, pp. 54–62, 2019.

[2] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal, "Rearchitecting linux storage stack for μs latency and high throughput," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021, pp. 113–128.

[3] A. Lerner and P. Bonnet, "Not your grandpa's ssd: The era of co-designed storage devices," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2852–2858.

[4] S. Yu, Z. Sha, C. Tang, Z. Cai, P. Tang, M. Huang, J. Li, and J. Liao, "Adaptive dram cache division for computational solid-state drives," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.

[5] S. Liang, Y. Wang, Y. Lu, Z. Yang, H. Li, and X. Li, "Cognitive ssd: A deep learning engine for in-storage data retrieval," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 395–410.

[6] S. Kim, Y. Jin, G. Sohn, J. Bae, T. J. Ham, and J. W. Lee, "Behemoth: a flash-centric training accelerator for extreme-scale dnns," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 371–385.

[7] Y. Lee, J. Chung, and M. Rhu, "Smartsage: training large-scale graph neural networks using in-storage processing architectures," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 932–945.

[8] K. Lee, I. Jo, J. Ahn, H. Lee, H. Lee, W. Sul, and H. Jung, "Deploying computational storage for htap dbmss takes more than just computation offloading," *Proceedings of the VLDB Endowment*, vol. 16, no. 6, pp. 1480–1493, 2023.

[9] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A user-programmablessd," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 67–80.

[10] G. Koo, K. K. Matam, T. I, H. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram, "Summarizer: trading communication with computing near storage," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 219–231.

[11] Z. Ruan, T. He, and J. Cong, “Insider: Designing in-storage computing system for emerging high-performance drive,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 379–394.

[12] R. Schmid, M. Plauth, L. Wenzel, F. Eberhardt, and A. Polze, “Accessible near-storage computing with fpgas,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–12.

[13] Z. Yang, Y. Lu, X. Liao, Y. Chen, J. Li, S. He, and J. Shu, “λ-io: A unified io stack for computational storage,” in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 347–362.

[14] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, “Near-data processing: Insights from a micro-46 workshop,” *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.

[15] Y. Huang, N. Guan, S. Bai, T.-w. Kuo, and C. J. Xue, “Serico: Scheduling real-time i/o requests in computational storage drives,” in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[16] A. Gupta, Y. Kim, and B. Urgaonkar, “Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings,” *Acm Sigplan Notices*, vol. 44, no. 3, pp. 229–240, 2009.

[17] Y. Luo, Y. Cai, S. Ghose, J. Choi, and O. Mutlu, “Warm: Improving nand flash memory lifetime with write-hotness aware retention management,” in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2015, pp. 1–14.

[18] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, “Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds,” *ACM Transactions on Storage (TOS)*, vol. 13, no. 3, pp. 1–26, 2017.

[19] “Daisyplus openssd,” https://www.crz-tech.com/crz/article/DaisyPlus.

[20] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, “The case of femu: Cheap, accurate, scalable and extensible flash emulator,” in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 83–90.

[21] “Stanford large network dataset collection,” https://snap.stanford.edu/data.