Revisiting Computational Storage for Data Integrity and Security

Chao Shi[†], Anthony Manschula[†], Tabassum Mahmud[†], Zeren Yang[§], Mai Zheng[†]
Yong Chen[‡], Jim Wayda[‡], Matthew Wolf[‡], Byungwoo Bang[‡]
[†]Iowa State University [§]University of Wisconsin-Madison [‡]Samsung

I. Introduction

The idea of computational storage device (CSD) has come a long way since at least 1990s [1], [2]. By embedding computing resources within storage devices, CSDs could potentially offload computational tasks from CPUs and enable near-data processing (NDP), reducing data movements and/or energy consumption significantly. While the initial hard-disk-based CSDs suffer from severe limitations in terms of on-drive resources, programmability, etc., the storage market has witnessed the commercialization of solid-state-drive (SSD) based CSDs (e.g., Samsung SmartSSD [3], ScaleFlux CSDs [4]) recently, which has enabled CSD-based optimizations for a variety of application scenarios (e.g., [5], [6], [7]).

Nevertheless, existing CSD research efforts mainly focus on performance acceleration of regular operations, leaving the potentials on system reliability/security largely unexplored. In this work, we attempt to bridge the gap. We revisit the classic idea of CSDs from a new angle: Can we leverage CSD to improve data integrity and/or security? To answer the question, we look into three representative I/O-intensive reliability/security techniques for data protection, and explore their similarities and potentials for CSD-based optimizations:

- *Fault Injection (FI)* is an indispensable method for testing the failure recovery of various storage systems (e.g., [8], [9], [10], [11], [12], [13], [14], [15]). We observe that the core operations of FI typically involve *intercepting I/O blocks* at certain software layer (e.g., kernel block layer [8], FUSE [12], drivers [10], [13], [14]) to implement the functionality. A CSD-based FI solution could potentially achieve similar I/O interception and manipulation at the bottom of the storage stack (i.e., device) to enable full-stack testing with high fidelity.
- *Erasure Coding (EC)* is an essential fault-tolerance mechanism for modern distributed storage systems (DSS) (e.g., Ceph [16], HDFS [17]). We observe that the core operations of EC involve *matrix multiplications* for encoding/decoding. In particular, locally repairable codes (LRC) [18] have been proposed to reduce the network and/or storage I/O cost by leveraging local parities, which could potentially benefit from FPGA-based optimization with a small set of collaborative CSDs.
- *Ransomware Detection & Recovery (RDR)* is increasingly important for protecting user data as ransomware has grown to a national security threat recently [19]. We observe that one major category of RDR solutions rely on SSDs [20], [21], [22], [23], [24], [25], [26] or hypervisor [27] to achieve *I/O pattern monitoring* for ransomware detection and *intra-device data movement* for data recovery, both of which aligns well with CSD characteristics. A CSD-based RDR could potentially achieve higher flexibility (compared to regular SSD-based RDR) and efficiency (compared to hypervisor-based RDR).

Based on the key observations above, we design a generic SmartSSD CSD library called `CSDGuard` to serve as a building block for constructing CSD-optimized reliability/security solutions. The library follows the Computational Storage Architecture Programming Model[28] to cover the core operations (e.g., host-device buffer management, I/O interception and monitoring, multi-dimensional array multiplication) of representative FI, EC, and RDR algorithms. Moreover, it provides a simple set of APIs to abstract away unnecessary CSD internals and support controlling data and metadata operations between host and CSDs with flexible configurability.

To demonstrate the potential of such a solution, we build a prototype of `CSDGuard` based on the Samsung SmartSSD platform [3]. The prototype leverages the peer-to-peer (P2P) transfer between NVMe flash storage and on-drive FPGA to minimize data communication between the host and the CSD, and applies a set of directive-based optimizations (e.g., `HLS INTERFACE`, `HLS ARRAY_PARTITION`, `HLS UNROLL`) to make full use of the massive parallelism of FPGA and thus achieve efficient near data processing. Our preliminary results are promising: Measuring the execution time of our library with directive-based optimizations applied, the overall latency was successfully reduced up to 70% across several experimental data sizes (e.g., the tested matrix size ranges from 384x384 to 2048x2048). With regard to P2P data transfer time, we observed similar performance to the conventional software-based data transfer approach between the CSD and host device. We believe we may be incurring some additional overhead in the system calls, which may lead to the behavior that we observed. We plan to extend the preliminary prototype to cover different use cases (e.g., FI, EC, and RDR) and evaluate with realistic systems (e.g., Ceph/HDFS with EC configuration) and datasets (e.g., VirusTotal) to fully demonstrate the potentials of CSD for data protection.

In order to compare the effectiveness of our design to a traditional CPU-based approach, the team collected information on data transfer and multiplication algorithm execution

times. The test setup for both the CPU and SmartSSD-based approaches are as follows: For both approaches, two input square matrices of a user-defined size are randomly generated on the host device and written to the NVMe flash on the SmartSSD. Figure 1 shows the flow of data when using the SmartSSD-based approach.

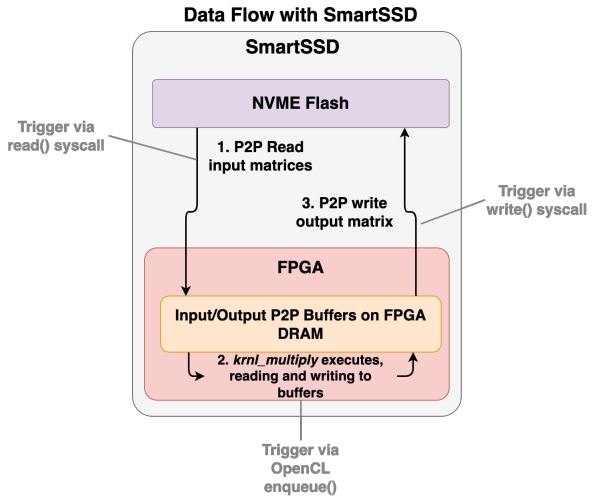

Fig. 1: *Operation of the Samsung SmartSSD-based approach with P2P.*

For the SmartSSD-based approach, performance measurement was done via two methods. As it can be seen in Figure 2, while data movement is entirely contained on the SmartSSD, the process still needs to be initiated by a read or write system call from the host device. In order to obtain the amount of time required for the data transfers to complete, the `high_resolution_clock()` function within the C++ *chrono* library was utilized, taking the current time before the read/write function was called, the current time after the call returns, and subtracting the two. To obtain accelerator kernel execution time, OpenCL Events were used. The `getProfilingInfo()` function provided by the Event class allows us the ability to extract performance profiling information from tasks that are queued, such as a hardware kernel. In this instance, we utilize the kernel start and end properties, subtracting in a manner as with the data transfer times.

The CPU-based approach follows a similar methodology, with the exception of the processing kernel. For a one-to-one comparison, the C code used to synthesize the hardware kernel was directly ported to the host device. The matrix computation code remained identical, with minor modifications (not impacting functionality) to read/write to the local buffers allocated on host memory as opposed to the FPGA DRAM. Performance measurement for the CPU-based solution was performed in a similar fashion to the SmartSSD-based solution. In this instance, we exclusively use the C++ *chrono* library to measure start and end times of data transfers and matrix processing. The behavior of the CPU-based system is outlined in Figure 2 below.

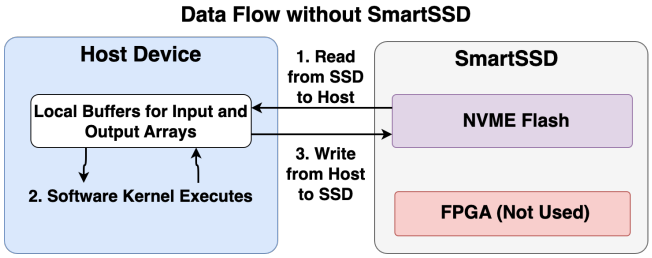

Fig. 2: *Operation of the system using the CPU-based approach.*

The system was tested with three different data input matrix sizes - 576KiB, 4MiB, and 9.2MiB. The first of the three sizes was chosen for a point of comparison with the unoptimized design, as that was the previous maximum size the kernel could handle. 4MiB (1024x1024 unsigned integer matrix) was chosen as an intermediate size during further testing in anticipation of the next size being a 16MiB (2048x2048 matrix). However, for reasons discussed in section 3.3, this was not possible, so we landed on a maximum of about 9.2MiB, or a 1536x1536 matrix. To ensure consistency in the execution time data that was collected for both approaches, 2000 consecutive runs of data reading and writing were made each test and 50 consecutive kernel runs were performed. The reasoning behind the lower number of kernel repetitions versus read/write repetitions is that the processing aspect exhibited much less run-to-run inconsistency when only one repetition was performed, allowing us to save time testing while still giving us confidence that the result was accurate.

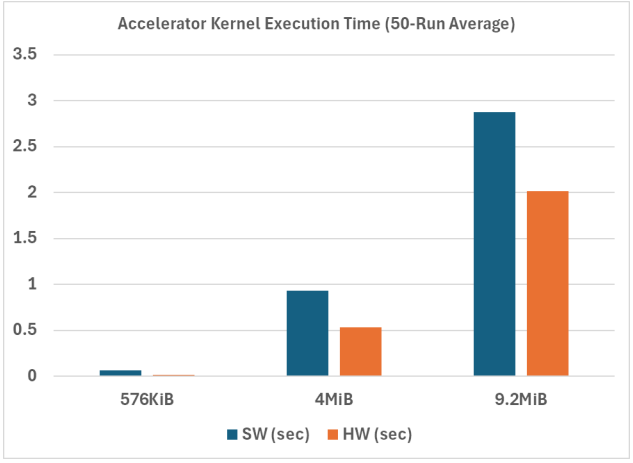

Fig. 3: *Kernel processing time for each data size, measured in seconds.*

We observed that the execution time of both the hardware and software kernel grew nonlinearly as the input data size scaled up. The software kernel required from 0.062 seconds

to complete at the 576KiB input size to as much as 2.876 seconds to complete at the 9.2MiB input size. The hardware kernel performed much better than the software kernel across the board, seeing over 3x speedup at the 576KiB level, down to about a 1.4x speedup at the 9.2MiB level. As a whole, the design showed significant improvement versus previous iterations. Compared to the midterm implementation of the accelerator, which required over 200ms to compute an output from 576KiB of input data in hardware, the new implementation completes in just 18ms. It is also worth considering that the results of the software implementation represent the performance of high-end server hardware, meaning that the FPGA implementation would likely pull further ahead when compared to a system with more pedestrian components.

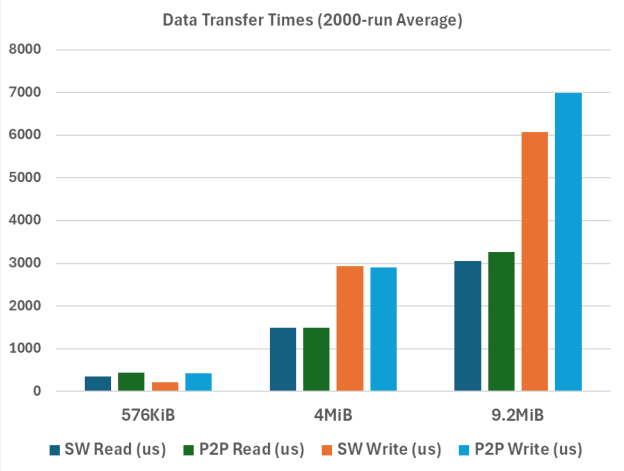

Fig. 4: *Time to complete data transfers of 576KiB, 4MiB, and 9.2MiB, measured in microseconds.*

Data transfer times tell a slightly different story. For one, the scaling as data size increases appears to be mostly linearly correlated. Furthermore, the transfer times for the CPU-based approach and the HW-based approach are largely the same across data sizes. Additionally, the run-to-run variance was sometimes quite large - taking a 2000-run average smooths the data for the most part, however some outliers like the 9.2MiB P2P write still show odd variance. This data will be discussed in more detail in section 3.3. The plot of end-to-end execution time of the software and hardware routines shown in Figure 6 has a nearly identical appearance to the kernel execution time chart. This is because data transfer times account for only a small portion of the overall latency, being around only a couple percent on average across all runs of hardware and software implementations.

An important aspect of the design to consider is the characteristics of the underlying hardware that is synthesized by Vitis HLS. As mentioned, the more performant version of our accelerator design utilizes loop unrolling; Loop unrolling creates many copies of the loop body for different iterations of the loop, allowing them to execute concurrently and (in theory) reducing the amount of time spent processing the loop. However, a critical downside to such an approach is

Fig. 5: *End-to-End program execution time, including kernel and data transfer times, measured in seconds.*

that the complexity and physical size of the synthesized design increases substantially, which has a negative impact on maximum clock speed. The team observed this effect firsthand: Kernels with larger input array sizes (meaning more loop iterations to fully compute the output matrix) led to a substantial reduction in final clock speed (see Figure 7). This reduction in clock speed likely contributes to the HW solution's observed decline from a 3x advantage to only 1.4x.

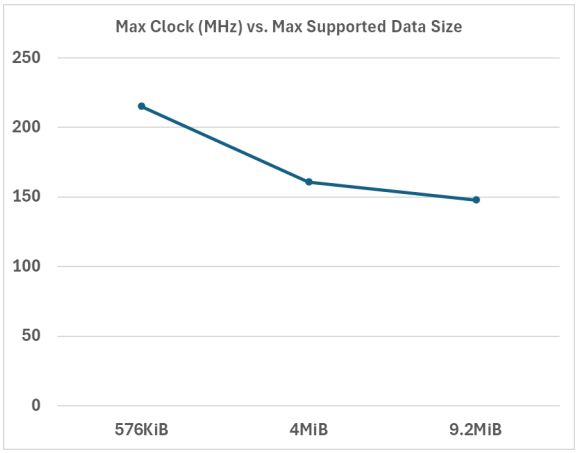

Fig. 6: *Max achievable clock speed compared to maximum supported input data size, as reported by the synthesis tool.*

Acknowledgements: This work was supported in part by National Science Foundation (NSF) under grands CNS-1855565 and CNS-1943204, and a Global Research Outreach (GRO) Award from Samsung (2022).

References

[1] A. Acharya, M. Uysal, and J. Saltz, "Active disks: programming model, algorithms and evaluation," *SIGOPS Oper. Syst. Rev.*, vol. 32, p. 81–91, oct 1998.

[2] E. Riedel, G. A. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia," in *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, (San Francisco, CA, USA), p. 62–73, Morgan Kaufmann Publishers Inc., 1998.

[3] AMD Xilinx, "Samsung smartssd." https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html

[4] L. Liu, H. Xu, Z. Niu, J. Li, W. Zhang, P. Wang, J. Li, J. C. Xue, and C. Wang, "Scaleflux: Efficient stateful scaling in nfv," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4801–4817, 2022.

[5] W. Cao, Y. Zhang, X. Yang, F. Li, S. Wang, Q. Hu, X. Cheng, Z. Chen, Z. Liu, J. Fang, B. Wang, Y. Wang, H. Sun, Z. Yang, Z. Cheng, S. Chen, J. Wu, W. Hu, J. Zhao, Y. Gao, S. Cai, Y. Zhang, and J. Tong, "Polardb serverless: A cloud native database for disaggregated data centers," in *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, (New York, NY, USA), p. 2477–2489, Association for Computing Machinery, 2021.

[6] Y. Qiao, X. Chen, N. Zheng, J. Li, Y. Liu, and T. Zhang, "Closing the b+-tree vs.{LSM-tree} write amplification gap on modern storage hardware with built-in transparent compression," in *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pp. 69–82, 2022.

[7] H. Kim, H. Y. Yeom, and H. Sung, "Understanding the performance characteristics of computational storage drives: A case study with smartssd," *Electronics*, vol. 10, no. 21, p. 2617, 2021.

[8] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Iron file systems," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, (New York, NY, USA), p. 206–220, Association for Computing Machinery, 2005.

[9] O. R. Gatla, M. Hameed, M. Zheng, V. Dubeyko, A. Manzanares, F. Blagojević, C. Guyot, and R. Mateescu, "Towards robust file system checkers," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[10] O. R. Gatla, M. Zheng, M. Hameed, V. Dubeyko, A. Manzanares, F. Blagojevic, C. Guyot, and R. Mateescu, "Towards robust file system checkers," *ACM Trans. Storage*, vol. 14, dec 2018.

[11] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh, "Torturing Databases for Fun and Profit," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 449–464, 2014.

[12] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redundancy does not imply fault tolerance: analysis of distributed storage reactions to single errors and corruptions," in *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, (USA), p. 149–165, USENIX Association, 2017.

[13] J. Cao, O. R. Gatla, M. Zheng, D. Dai, V. Eswarappa, Y. Mu, and Y. Chen, "Pfault: A general framework for analyzing the reliability of high-performance parallel file systems," in *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, (New York, NY, USA), p. 1–11, Association for Computing Machinery, 2018.

[14] R. Han, O. R. Gatla, M. Zheng, J. Cao, D. Zhang, D. Dai, Y. Chen, and J. Cook, "A study of failure recovery and logging of high-performance parallel file systems," *ACM Trans. Storage*, vol. 18, apr 2022.

[15] R. Han, C. Shi, T. Mahmud, Z. Yang, V. Esaulov, L. Wan, Y. Chen, J. Wayda, M. Wolf, and M. Zheng, "Revisiting erasure codes: A configuration perspective," in *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '24, (New York, NY, USA), p. 93–100, Association for Computing Machinery, 2024.

[16] Ceph, "(accessed april 3, 2024)," https://ceph.com/en/.

[17] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, 2010.

[18] S. Kadekodi, S. Silas, D. Clausen, and A. Merchant, "Practical design considerations for wide locally recoverable codes (lrcs)," *ACM Trans. Storage*, vol. 19, nov 2023.

[19] B. Ma, Y. Yang, J. Li, F. Zhang, W. Shen, Y. Zhou, and J. Ma, "Travelling the hypervisor and ssd: A tag-based approach against crypto ransomware with fine-grained data recovery," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, (New York, NY, USA), p. 341–355, Association for Computing Machinery, 2023.

[20] S. Baek, Y. Jung, D. Mohaisen, S. Lee, and D. Nyang, "Ssd-assisted ransomware detection and data recovery techniques," *IEEE Transactions on Computers*, vol. 70, no. 10, pp. 1762–1776, 2021.

[21] S. Baek, Y. Jung, A. Mohaisen, S. Lee, and D. Nyang, "Ssd-insider: Internal defense of solid-state drive against ransomware with perfect data recovery," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 875–884, 2018.

[22] J. Huang, J. Xu, X. Xing, P. Liu, and M. K. Qureshi, "Flashguard: Leveraging intrinsic flash properties to defend against encryption ransomware," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, (New York, NY, USA), p. 2231–2244, Association for Computing Machinery, 2017.

[23] D. Min, Y. Ko, R. Walker, J. Lee, and Y. Kim, "A content-based ransomware detection and backup solid-state drive for ransomware defense," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 7, pp. 2038–2051, 2022.

[24] J. Park, Y. Jung, J. Won, M. Kang, S. Lee, and J. Kim, "Ransomblocker: a low-overhead ransomware-proof ssd," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2019.

[25] B. Reidys, P. Liu, and J. Huang, "Rssd: defend against ransomware with hardware-isolated network-storage codesign and post-attack analysis," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, (New York, NY, USA), p. 726–739, Association for Computing Machinery, 2022.

[26] P. Wang, S. Jia, B. Chen, L. Xia, and P. Liu, "Mimosaftl: Adding secure and practical ransomware defense strategy to flash translation layer," in *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, CODASPY '19, (New York, NY, USA), p. 327–338, Association for Computing Machinery, 2019.

[27] F. Tang, B. Ma, J. Li, F. Zhang, J. Su, and J. Ma, "Ransomspector: An introspection-based approach to detect crypto ransomware," *Computers & Security*, vol. 97, p. 101997, 2020.

[28] SNIA, "Computational Storage Architecture and Programming Model v1.0." https://www.snia.org/sites/default/files/technical-work/computational/release/SNIA-Computational-Storage-Architecture-and-Programming-Model-1.0.pdf. Accessed: 2022-08-30.