ASSASIN: Architecture Support for Stream Computing to Accelerate Computational Storage

Chen Zou* and Andrew A. Chien*†
* *Department of Computer Science, University of Chicago*
† *Mathematics and Computer Science Division, Argonne National Laboratory*
Email: {chenzou@, achien@cs.}uchicago.edu

***Abstract*—Computational storage adds computing to storage devices, providing potential benefits in offload, data-reduction, and lower energy. Successful computational SSD architectures should match growing flash bandwidth, which in turn requires high SSD DRAM memory bandwidth. This creates a *memory wall* scaling problem, resulting from SSDs' stringent power and cost constraints.**

A survey of recent computational SSD research shows that many computational storage offloads are suited to stream computing. To exploit this opportunity, we propose a novel general-purpose computational SSD and core architecture, called ASSASIN (Architecture Support for Stream computing to Accelerate computatIoNal Storage). ASSASIN provides a unified set of compute engines between SSD DRAM and the flash array. This eliminates the SSD DRAM bottleneck by enabling direct computing on flash data streams. ASSASIN further employs a crossbar to achieve performance even when flash data layout is uneven and preserve independence for page layout decisions in the flash translation layer. With stream buffers and scratchpad memories, ASSASIN core's memory hierarchy and instruction set extensions provide superior low-latency access at low-power and effectively keep streaming flash data out of the in-SSD cache-DRAM memory hierarchy, thereby solving the memory wall.

Evaluation shows that ASSASIN delivers 1.5x - 2.4x speedup for offloaded functions compared to state-of-the-art computational SSD architectures. Further, ASSASIN's streaming approach yields 2.0x power efficiency and 3.2x area efficiency improvement. And these performance benefits at the level of computational SSDs translate to 1.1x - 1.5x end-to-end speedups on data analytics workloads.

***Keywords*-computational storage, ssd, stream computing, memory hierarhcy, memory wall, general-purpose**

I. Introduction

The extraordinary scaling of NAND flash [1] drives rapid increase in capacity and bandwidth of SSDs. Leading enterprise SSDs can reach 6.95 GB/s and 900K IOPS [2], and even consumer SSDs exceed 7 GB/s and 690K IOPS [3]. These performance advances make SSDs essential to high performance data-intensive applications in data centers. This shifting performance balance in data centers is driving a rethink of the relationship between storage and compute to ensure that compute can keep up with the storage scaling.

One approach is disaggregated storage, as shown in Figure 1 (center). This approach allows separate scaling of compute to match the improving flash bandwidths. However, this approach drastically increases the compute requirements for both packet and data processing. Further, the bandwidth requirement of the interconnect between compute and disaggregated storage must also scale with the flash bandwidth, increasing the chance of interconnect becoming a hot spot in the system. Moreover, disaggregated storage does not address the aggressive scaling of the flash array inside a SSD to expose its growing bandwidth currently bottlenecked by the SSD interface [4].

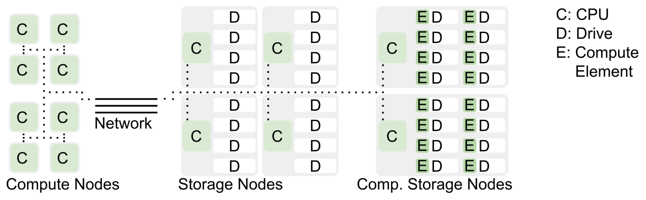


Figure 1: Computational storage in data centers

As a result, recent innovation [5], [6] and research [4], [7]–[16] explore computational SSDs [17], building on work dating back to the 1990s [18], [19]. Computational SSD embeds computation capabilities (E) in SSDs (D) as shown in Figure 1 (right), enabling early data filtering and computation scaling to address the SSD interface, interconnect and CPU compute bottlenecks.

However, matching computing performance to growing flash bandwidth inside a computational SSD is challenging. Specifically, performance must be delivered within the stringent power constraints of an SSD device (less than 5W [3]). In Section III, we show that compute performance of the typical computational SSD architecture shared by state-of-the-art general-purpose systems [8], [10], [11], [13], [20] is limited by the SSD DRAM, akin to the *memory wall* in CPUs. As a result, we explore a novel computational SSD architecture that addresses the memory wall problem without the loss of generality (independence of flash data placement and management to support both conventional SSD read/write requests as well as diverse computational storage functions concurrently). This novel general-purpose architecture delivers low-cost, low-power and high-performance computing inside a SSD.

We start with a broad study of storage offloads, searching for shared workload properties that could be exploited. The study shows that many computational storage functions can be implemented in a stream computing fashion. To exploit this insight, we propose ASSASIN (Architecture Support for Stream computing to Accelerate computatIoNal Storage), a novel general-purpose computational SSD architecture that enables computation directly on data that are streaming in and out of flash channels. ASSASIN includes new memory structures and operations (as an instruction set extension) that increase computing efficiency on streaming storage data.

Specific contributions of the paper include:

- Analysis of 14 computational storage functions from 22 research studies, showing that most computational storage functions are feasible with stream computing.
- a novel SSD architecture – ASSASIN – that provides unified computing engines between the flash array and SSD DRAM, eliminating the SSD DRAM bottleneck by enabling general-purpose computing on flash data streams, while preserving the independence of the flash translation layer to determine SSD data layout and page mapping, and thus the generality.
- a novel core architecture – ASSASIN Core – that efficiently processes flash data chunks, using stream buffers and a scratchpad memory. The mechanisms and instruction set provide superior low-latency access at low-power and enable programs to keep the flash data out of the SSD cache-DRAM hierarchy.

The paper is organized as follows. We cover the background on SSD architecture and NAND flash in Section II. Our motivation is then detailed in Section III, where the *memory wall* problem inside computational SSDs is raised. We conduct workload studies in Section IV for shared properties to address this memory wall. Our ASSASIN SSD architecture and ASSASIN core architecture are discussed in Section V, which we further evaluate in Section VI. Related work is discussed in Section VII. We summarize and identify directions for future work in Section VIII.

II. Background

A. Solid-state drive architecture

As shown in Figure 2, an SSD is composed of several flash chips organized in multiple channels, a DRAM chip and a controller chip. The controller chip comprises a firmware processor, a host interface controller, a DRAM controller and one flash controller for each flash channel.

The firmware processor runs the flash translation layer (FTL). FTL maintains the mapping between logical block addresses and physical block addresses. A physical block address specifies a physical location composed of a flash chip ID and the specific location inside the chip. FTL would consider both flash's read/write/erase granularity and the wear-leveling policy.

The flash chips organized in multiple channels are managed by the flash controllers, one per channel. The firmware

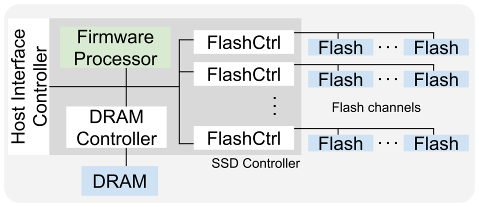


Figure 2: SSD architecture diagram

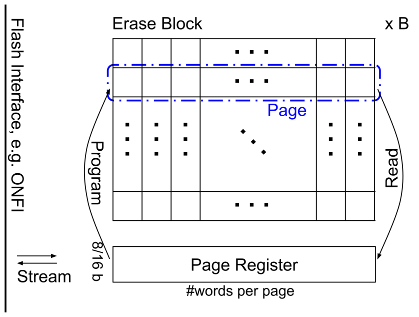


Figure 3: NAND flash

processor would issue requests to the flash controller in charge to access a specific flash chip. And the flash controller would respond with the required flash page if needed. The flash chips of the same channel share the same bus but operate independently. It is firmware and flash controllers' responsibility to exploit the operation interleaving opportunities among the flash chips sharing the same channel, analogous to the notion of ranks and rank-level parallelism in the memory system.

There is a DRAM chip in high-performance SSDs serving as a buffer to store both page data relevant to recent requests from/to flash controllers and request queues between the firmware and flash controllers. It also buffers FTL-related data structures for the firmware.

The host interface controller implements the storage protocol the SSD would use to communicate with the host, e.g. SATA, NVMe, etc. The firmware running on the firmware processor would pull requests and send responses from/to the host via this host interface controller.

B. NAND flash

We look deeper into the underlying NAND flash internals and flash interface. This is detailed in Figure 3. NAND flash is composed of multiple (B in the figure) erase blocks. Each block is an array of floating-gate MOSFET transistors sharing the substrate. The transistors on the same row sharing a selection line form the smallest read or program granularity which is called an flash page.

The NAND flash interacts with external entities with the page register through the flash interface. During a page read process, the selected row is first loaded into the page register, and the page streams out in a word-by-word (either 8b or 16b) fashion. During a page write process, data stream into the page register first, and the page register is used to program a specific row in the array. ONFI [21] is the standardization effort on the flash interface. The newest version as of writing is 5.0 which supports up to 2400 MT/s at the interface.

III. Motivation and the Problem

A. A Motivating Example

Let us consider an exemplar function, Filter, offloaded to computational storage from data analytics workload to understand the performance characteristics inside the state-of-the-art general-purpose computational SSD architecture shown in Figure 4, which is employed in multiple computational SSD studies [8], [11], [13], [15], [22]–[24].

The function filters tuples based on given predicates on certain fields. Tuples come from the database (to be specific, TPC-H lineitem table) stored in the SSD flash array of which the schema is known and no parsing is needed. This Filter function features early data reduction benefits when carried out with computational SSD because unselected data would not come out of the SSD interface. Further, the function features tuple-level parallelism, such that it is easy to exploit scaled-out compute engines inside SSD for high performance. Each engine could process tuples in a small batch of flash pages. Thus, this Filter function is suitable for offload to computational SSDs, and is considered in most computational storage studies (see Table I).

In the state-of-the-art general-purpose computational SSD architecture as shown in Figure 4, offloaded functions execute on compute engines (embedded-class cores). Data is first staged in the SSD DRAM, and then accessed by compute engines through the cache-DRAM memory hierarchy for processing. With a 1GHz in-order RISC-V scalar core on top of a 32KB 8-way L1 data cache and a 256KB 16-way L2 cache, the offloaded Filter function runs at 0.63 GB/s. Simulation done with Gem5 [25] with cache performance measured through Cacti [26] @ 14nm.

Although the Filter function is light on computing intensity, the achieved performance is far from the bandwidth of a flash channel (1.6GB/s or 3.2 GB/s depending on channel width as defined in ONFI 4.2 [21]). Digging deeper, we find that the performance is hindered by memory access stalls, as shown in the cycle decomposition detailed in Figure 5. Even if we assume the L1 cache is tiny (no extra delay for memory accesses) but perfect (no cache misses except compulsory). Compulsory misses and DRAM accesses would still slow down the performance by three times.

At the SSD level, let us consider the setting of eight 8-bit flash channels delivering 12.8GB/s to maximally utilize the current and future NVMe over PCIe storage interface (4 lanes of PCIe 4.0 peak at 8GB/s). The memory bandwidth requirement of the SSD DRAM is at least 25.6GB/s (read pages from flash controllers into DRAM, and compute engines read pages from DRAM to perform in-SSD computing). This is shown with blue arrows in Figure 4). The full requirement will be higher, as there is additional traffic for writing the in-SSD computing results or firmware DRAM access needs. These requirements exceed the capabilities of LPDDR4 DRAM used in current SSD products [3], and even exceeds a DDR4 DRAM of 16GB/s sustained bandwidth.

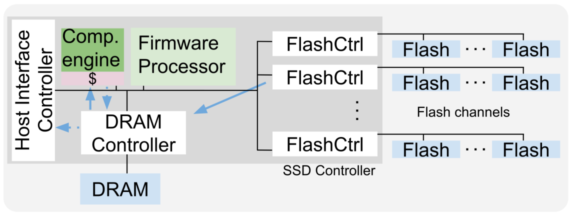


Figure 4: State-of-the-art computational storage architecture

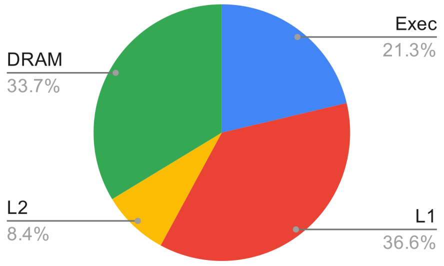


Figure 5: Cycle decomposition for 'Filter'

B. The Problem

Our example illustrates the *memory wall* problem inside a computational SSD. State-of-the-art computational SSDs read data from the flash array into the SSD DRAM. And compute elements then process these data on DRAM through the cache. This makes SSD DRAM bandwidth a critical performance limit for hosting both compute and flash traffics. As flash bandwidth scales, so must the computation, producing increasing demands on DRAM bandwidth and creating a hot spot in the SSD architecture.

CPU solutions to the *memory wall*, including caching and increase on memory parallelism (multi-channel or HBM), do not readily apply. Caches do not work well for streaming data due to low reuse. Increased memory parallelism is both expensive and high-power. These costly CPU approaches are not viable in the extremely cost and power-sensitive SSD storage space (backbone of modern data centers).

The computing and memory hierarchy architecture inside computational SSDs should be lightweight but effective. It should enable compute engines inside to access data with low latency, low power, but high bandwidth. It should also feature moderate consumption of silicon resources. We dive into the workloads for insights that may help us resolve the *memory wall* problem in computational SSDs.

	File system					Database					Other			
	Cryptography	Compress	Deduplicate	Erasure Coding	Replicate	Filter	Select	Parse	Statistics	Write-ahead Log Replay	Transpose	Statistical Modeling	Neural Networks	Graph Analytics
Access [22]	x	x				x								
ActiveFlash [15]		x				x			x		x	x		
Aurora [6]					x					x				
Azure [27]				x										
Biscuit [13]				x		x								
BlockIF [23]				x										
Caribou [14]		x	x		x	x	x							
CIDR [28]		x	x											
DedupInSSD [29]			x											
DeepStore [30]													x	
Glist [31]														x
Grafboost [32]														x
Ibex [7]						x	x	x						
IceClave [20]	x					x	x		x					
Insider [24]		x				x	x		x			x		
Lepton [33]		x												
MithriLog [34]						x	x	x	x					
Query [35]						x	x							
Skyhook [36]		x				x	x		x					
Summarizer [11]						x	x		x					
Thrifty [37]												x	x	
YourSQL [8]						x	x	x						

Table I: Functions from different application domains proposed for computational storage

IV. Understanding the workload

A. *Computational storage offload spectrum*

We surveyed the research literature considering offloading computations either into storage systems, along the storage links (network/PCIe) or inside storage devices [6]–[8], [11], [13]–[15], [20], [22]–[24], [27]–[38]. We extract the specific offloaded functions from each system and summarize them in Table I. Offload functions are shown as columns and system/literature names as rows. As we found most functions considered for computational storage offload are from the file system or database domain, we further cluster the relevant functions (columns) together. The table collectively represents a cross-section of system research on computational storage.

B. *Function structure and memory access requirements*

Dwelling on computational storage functions, we find that their implementations all feature streaming access to storage data and random access to function states.

Starting with file system functions, for Cryptography, the key schedule is usually determined at the beginning and thus suitable for storing as function states. Encryption or decryption would be applied to the input data or code blocks in a streaming fashion. For compression and decompression, besides the input data streaming in, indexes to the dynamic dictionary (i.e. recent history of the compressing or decompressing data) could be seen as function states.

Table II: Stream computing implementation of computational storage functions

	Streaming	**Function States**
Cryptography	Data blocks / Code blocks	Keys & GF table
(De)compress	Data and history	Dictionary indexes
Deduplicate	Data blocks	Block metadata
Erasure coding	Data blocks / Code blocks	Galois Field (GF) table
Replicate	Data & Replicates	–
Filter	Tuples	Flags
Select	Tuples	–
Parse	Tuples	State machines
Statistics	Tuples	Accumulators
NN Training	Training data	Model parameters
NN Inference	Inference input	Model parameters
Graph Analysis	Edge list / Vertex list	Statistics

Different implementations all have an explicit upper bound on the history size, which also limits the size of an additional suffix tree or a hash table for indexing this dynamic dictionary. Deduplication is similar to compression except the dictionary being metadata on seen blocks. For erasure coding, it reads in multiple streams of data blocks and generates extra coded blocks through various Galois field operations before streaming out. There are no states but a Galois field multiplication lookup table used across erasure coding operations for different blocks.

For database functions like Filter, Select, Parse, these workloads feature a highly parallel nature where computation is applied to each row of data independently. As a result, aside from temporaries and streaming input/output of table data and results, there are no function states for state transfers. It is similar for the Statistics function which generates statistical summaries from data tuples, it only needs additional accumulators as the function states.

For neural network training and inference, it is sensible for either a general-purpose processor or an accelerator to keep weights of the model stationary in fast-and-close memory (e.g. scratchpads) and streaming in the inference or training data. And for graph analysis, we can also stream the edge list or vertex list while performing updates on the statistics kept in close memory. We summarized how functions are mapped to stream computing in Table II.

V. ASSASIN Design

Inspired by the shared workload property of streaming, we design ASSASIN to achieve efficient inline stream computing on data going in and out of the flash array. ASSASIN keeps these streams out of the cache-DRAM hierarchy, efficiently avoiding the SSDs' memory wall.

A. ASSASIN SSD: Stream computing between flash controllers and DRAM

The ASSASIN SSD architecture is shown in Figure 6. Contrast to a regular SSD architecture as shown in Figure 2, ASSASIN adds scalable stream computing cores (ASSASIN cores) on the SSD controller chip logically between the SSD DRAM and the flash array. ASSASIN cores carry out offloaded functions as inline stream computing on the data stream (denoted by the blue arrows in Figure 6) between SSD DRAM and the flash array. Note that flash controllers hide the electrical complexity of flash, such that ASSASIN cores can be implemented with standard CMOS VLSI methodologies.

Comparing to the state-of-the-art general-purpose computational SSD architecture as shown in Figure 4, where compute engines fetch storage data through a traditional cache-DRAM memory system only after data are first staged in SSD DRAM, ASSASIN saves at least half of the SSD DRAM traffic. Further, function offloads to computational storage often reduce data (read) or increment data (write), generating system benefit by decreasing traffic at the storage interface. ASSASIN allows these functions to be implemented between SSD DRAM and the flash array via ASSASIN cores, harvesting these traffic reduction benefits for the SSD DRAM as well. As a result, ASSASIN SSD architecture largely reduces SSD DRAM traffic and bandwidth requirement, addressing the memory bottleneck issue raised in Section III.

ASSASIN also differs significantly from proposed application-specific computational SSD architectures (Figure 7) that add specialized compute engines to each channel/controller or to each flash die [30], [37], [39]. ASSASIN has two key advantages: First, ASSASIN flexibly shares compute engines across the SSD with a

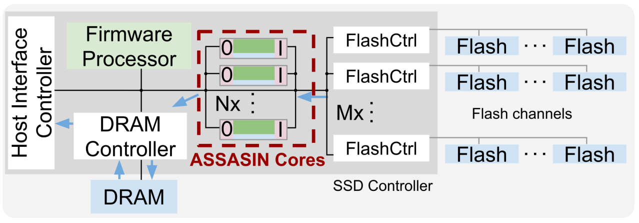


Figure 6: ASSASIN SSD (Compare with Figures 4 and 7)

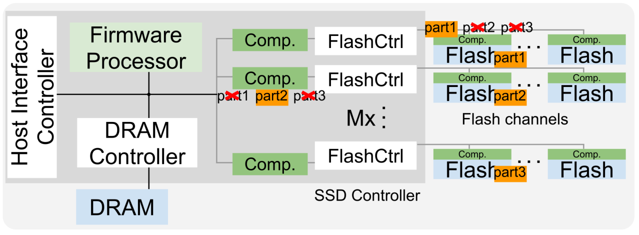


Figure 7: Application-specific computational storage proposals [30], [37], [39] bind acceleration to channels, and thus cannot flexibly share compute and compose data from across the flash array

crossbar interconnect, delivering robust performance even with uneven data distribution across channels. Second, ASSASIN's crossbar interconnect enables aggregating pages from different channels and presenting them to the compute engines. This allows FTL placement and management decisions to be completely independent (and thus, no customized FTL is required for ASSASIN). As a result, ASSASIN can support flexible interleaving of read/write requests that do not exploit computational storage with computational storage operations.

B. ASSASIN core: Efficient Streaming

An ASSASIN core is based on a general-purpose core (like the firmware processor) but extends it in following aspects.

Hybrid hierarchy for inline streaming. Each ASSASIN core employs a hybrid memory hierarchy consisting of input/output streambuffers, a scratchpad and a cache, as shown in Figure 8. Input and output streambuffers are for low-latency storage stream access. Scratchpad is tightly integrated with core pipeline and thus offers low-latency random access to function state. The cache which is further backed by SSD DRAM is for holding data structures larger than scratchpad's limit (i.e. a fallback capacity memory).

Stream buffer under a microscope. As drawn in Figure 8, a stream buffer can hold up to S streams. For each stream, there is a circular buffer with the capacity of P flash pages. Here, P and S are both microarchitecture parameters. There are two pointers on each circular buffer, Head and Tail, which are both control status registers (CSR) of an ASSASIN core. Head points to a core's current position on the stream. The word at the Head position could be easily prefetched into

Table III: Instruction set extension for stream access and management

Instruction	Description	Instruction format [31:0]
StreamLoad	Hangs if empty, i.e. IHead[rs1] == ITail[rs1]. Otherwise: R[rd] = IStream[rs1][IHead[rs1]], IHead[rs1] += width	unused[11:0] rs1[4:0] width[2:0] rd[4:0] opcode[6:0]
StreamStore	Hangs if full, i.e. OHead[rs1] + 8 > OTail[rs1]. Otherwise: OStream[rs1][OHead[rs1]] = R[rs2], OHead[rs1] += width	unused[11:5] rs2[4:0] rs1[4:0] width[2:0] unused[4:0] opcode[6:0]
ReadIStream	R[rd] = IStream[rs1][ITail[rs1] - R[rs2]]	unused[6:0] rs2[4:0] rs1[4:0] width[2:0] rd[4:0] opcode[6:0]
ReadOStream	R[rd] = OStream[rs1][OHead[rs1] - R[rs2]]	unused[6:0] rs2[4:0] rs1[4:0] width[2:0] rd[4:0] opcode[6:0]

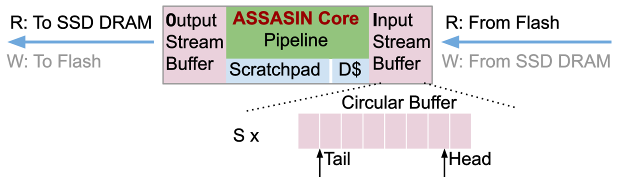


Figure 8: ASSASIN core

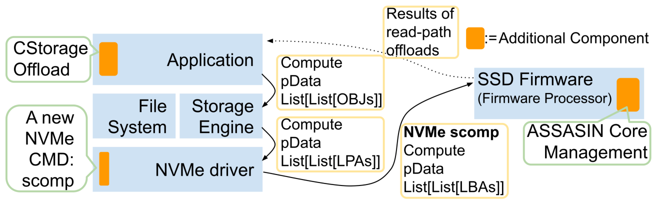

CStorage Offload
Application
Results of read-path offloads
:=Additional Component
Compute pData List[List[OBJs]]
SSD Firmware (Firmware Processor)
File System
Storage Engine
A new NVMe CMD: scomp
Compute pData List[List[LPAs]]
NVMe scomp Compute pData List[List[LBAs]]
ASSASIN Core Management
NVMe driver

Figure 9: Software stack extensions for ASSASIN

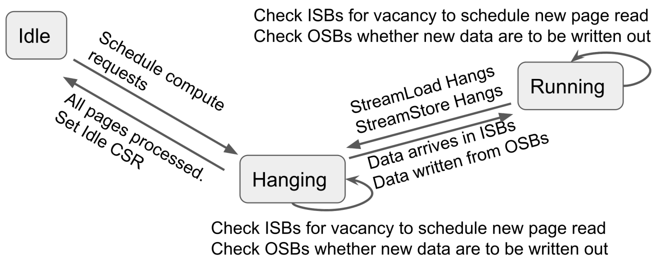

Idle
Schedule compute requests
All pages processed. Set Idle CSR
Check ISBs for vacancy to schedule new page read
Check OSBs whether new data are to be written out
Running
StreamLoad Hangs
StreamStore Hangs
Data arrives in ISBs
Data written from OSBs
Hanging
Check ISBs for vacancy to schedule new page read
Check OSBs whether new data are to be written out

Figure 10: ASSASIN core management FSM in firmware

the core pipeline, allowing low-latency access. The Tail CSR acts as a doorbell register to be used by the SSD firmware. The firmware would update (advance) this Tail register to let an ASSASIN core know that new data are fetched into the circular buffer, ready to be processed.

Instruction set extension for streaming. Besides the CSRs described in previous paragraphs, the ASSASIN core augments a general-purpose ISA (we use RISC-V [40] in evaluation) with additional instructions to access (and automatically manage) input and output streams, which we summarized in Table III. The first two instructions StreamLoad and StreamStore feature automatic stream pointer increments (i.e. Header CSR) based on the ‘width’ immediate, while the latter two have no effects on streambuffer pointers.

C. Flexible interconnect: scalable compute

We architect ASSASIN to be scalable with the flash array bandwidth through pooling cores at the SSD-level connected with a crossbar interconnect. This allows a flexible N:M pairing (as the ‘N’ and ‘M’ in Figure 6), between the ASSASIN cores and flash channels to scale out compute performance, as opposed to the fixed 1:1 pairing for channel-level compute engines [30], [37].

One may wonder whether ASSASIN is just shifting the traffic and bandwidth requirements from the SSD DRAM to the interconnect. The insight here is that the interconnect and stream buffers are streaming-oriented (small and restrictive thus allowing optimizations). They can be scaled to high bandwidth at much lower power when compared with the NV-DDR [21] facility that transfer pages from channels to the SSD DRAM (huge and random accesses). This advantage is the root of the system efficiency of ASSASIN. As a confirmation, a 16-bit per input/output lane, 16x16 crossbar is implemented as the interconnect in SystermVerilog. It easily closes timing at 2GHz with 14nm SAED library and is only one eighth of a core in silicon area (see Section VI-G).

D. ASSASIN Programming Model

Software stack extensions. Generally, computational SSD requests are specified in the form of ‘(compute, pData, List[List[LPA]])’. ‘compute’ represents a stream computing function, the specification of which is detailed in the third part of this subsection. ‘pData’ is a host pointer to either the input data for write-path computational SSD requests the results of which would be writing to storage, or the destination to store the results for read-path computational SSD requests. The 2-dimensional array specifies the logical page addresses (LPA) that the output stream(s) of a write-path offloaded function should be written to or that form the input stream(s) for a read-path function. And the size of the outer dimension corresponds to the number of input/output streams. A computational SSD request would be wrapped as a new NVMe command ‘scomp’, as shown in Figure 9.

If upper-layer applications would rather specify compute inputs in List[List[objects]], a storage engine would be responsible of transforming that in to List[List[LPA]]. This is the original responsibility a storage engine (a file system or a DBMS storage engine) would assume even without computational storage, so no changes are required here. Further, this is where task decomposition would take place to exploit the task-level parallelism through the multiple ASSASIN cores at the SSD-level. Large compute requests

Listing 1: ‘compute’ specified as a function

```
void compute(char* scratchpad) {
    while (true) { // An iteration processes one object, e.g. a tuple.
        char input = StreamLoad(0, 1); // InStreamId == 0, width == 1.
        char output;
        // Compute on input or fetch more data from input stream to produce output.
        // Maybe also use scratchpad in the process.
        // ...
        StreamStore(0, 1, output); // OutStreamId = 0, width = 1.
    }   // The loop ends when StreamLoad hangs, i.e. input stream is exhausted.
        // Firmware would reset ASSASIN core (PC & pipeline) before next compute request starts.
}
```

can be decomposed into multiple smaller ones with consistent splitting of each object/LPA stream.

ASSASIN core management in the firmware. Following the insights of control plane and data plane separation [24], the firmware processor (as shown in Figure 6) runs firmware (i.e. the control plane) independent of compute-oriented ISA/uArch innovations employed by ASSASIN cores. Compared to conventional SSDs, the firmware is extended with the capability of managing ASSASIN core resources. This includes schedule stream computing on any ASSASIN core and schedule page read and write to/from any input/output streambuffer (ISB/OSB).

As shown in Figure 10, the firmware periodically checks control status registers (Head/Tail for each ISB and OSB) of each ASSASIN core, performing state transitions (hanging avoids overflow) and schedules pages in and out streambuffers. This is where the construction of streams from pages at specified LPAs in computational storage requests (see the first part of this subsection) takes place. ASSASIN cores only process streams, without the need of knowing any flash array data layout or LPAs (preserving generality). The management process is similar to that of in-DRAM buffers which firmware originally assumes, thus no new challenges are involved.

Specify streaming ‘compute’. ‘compute’ needs to be written in a streaming fashion, as shown in Listing 1. It reads from the input data stream(s) using StreamLoad instructions, performs required computation and appends output to the output data stream(s) via StreamStore instructions.

For existing applications, offload functions will in general need to be rewritten in a streaming fashion with ASSASIN ISA extension instructions (wrapped in intrinsics for high-level programming languages). However, compiler support to automate this task is feasible via automatic streaming access pattern identifications [41].

VI. Evaluation

We simulate several computational storage kernels and full-system TPC-H data-analytics pipelines and compare ASSASIN against state-of-the-art SSD architectures.

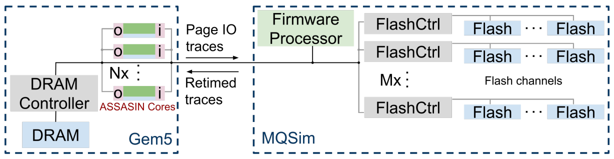
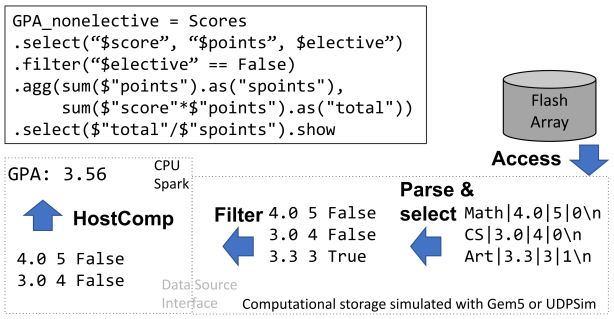


Figure 11: Hybrid simulation infra with Gem5 and MQSim



```
GPA_nonelective = Scores
.select(“$score”, “$points”, $elective”)
.filter(“$elective” == False)
.agg(sum($"points").as("spoints"),
     sum($"score"*$"points").as("total"))
.select($"total"/$"spoints").show
```

```
GPA: 3.56
```

```
4.0 5 False
3.0 4 False
```

```
4.0 5 False
3.0 4 False
3.3 3 True
```

```
Math|4.0|5|0\n
CS|3.0|4|0\n
Art|3.3|3|1\n
```

Figure 12: Compstor-accelerated analytics pipelines

A. Methodology

Configurations. We compare six computational SSDs that differ in compute engines and their integration in the SSD architecture, as summarized in Table IV. In other aspects the SSDs are similar, employing an 8-channel flash array (each channel has 1GB/s read/write performance), a 2GB LPDDR5 DRAM [3] (8GB/s effective bandwidth), and a PCIe Gen4 x4 host interface. In all cases, the host CPU driving the computational SSD is four-core eight-thread with 32GB main memory.

Compute engines in each SSD (except the UDP, see Table IV) are eight in-order scalar RISC-V cores with different memory hierarchies. RISC-V cores are selected because of the availability of open-source designs (we use ibex cores [44]) and toolchain [45]. **Baseline**, as drawn in Figure 4, represents state-of-the-art general-purpose computational SSD architecture [8], [11], [13], [15], [22]–[24], where compute engines fetch data from SSD DRAM before computing on it. The other variants (Prefetch, AssasinSp, AssasinSb and AssasinSb$) add varied memory hierarchies at each core. **Prefetch** adds a prefetcher from

Table IV: Configurations of in-SSD compute engines

	Data Source	ISA	#Cores	Frequency	MemArch per Core. 32KB L1I omitted
Baseline	DRAM (8GB/s)	RISCV32IM	8	1 GHz	L1D: 32KB, 8 way, 64B cache line L2: 256KB, 16 way, 64B cache line
UDP [42]	DRAM (8GB/s)	UDP ISA	8	1 GHz	256KB scratchpad
Prefetch	DRAM (8GB/s)	RISCV32IM	8	1 GHz	L1D: 32KB, 8 way, 64B cache line L2: 256KB, 16 way, 64B cache line DCPTPrefetcher [43] (best among Gem5 prefetchers)
AssasinSp	**S**cratch**p**ad	RISCV32IM	8	1 GHz	64KB scratchpad 64KB I + 64KB O ping-pong scratchpads
AssasinSb	**S**tream**b**uffer	RISCV32IM +Stream ISA	8	1 GHz	64KB scratchpad 64KB I + 64KB O streambuffer (S=8 P=2)
AssasinSb$	**S**tream**b**uffer + Cache	RISCV32IM +Stream ISA	8	1 GHz	64KB scratchpad, 32KB 8W L1D 64KB I + 64KB O streambuffer (S=8 P=2)

SSD DRAM for latency reduction. There are three ASSASIN variations. In **AssasinSp**, conventional **S**cratch**p**ads are used to double-buffer storage data in a 'ping-pong' fashion. Storage data is fetched from flash into the 'pong' scratchpad, bypassing the SSD DRAM while the compute engines process data already in the 'ping' scratchpad and vice versa. **AssasinSb** employs a **S**tream**b**uffer which not only enables direct computation on storage data streams bypassing SSD DRAM, but also automatically manages the stream pointers through the core-level stream ISA extension (Section V-B). **AssasinSb$** adds a data cache (backed by DRAM) for increased flexibility, with graceful degradation if the scratchpad size is not large enough for offload functions' needs of random access.

UDP [42], designed to accelerate data analytics, represents another dimension of computational SSD architectures [7], [14], [28], [30], [34], [39] where application- or domain-specific accelerator(s) are employed. A UDP lane computes using a private scratchpad. The firmware processor copies data to be processed from SSD DRAM into these scratchpads.

Simulation. We adopt a hybrid simulation methodology as shown in Figure 11. Gem5 [25] is employed to model different memory hierarchy configurations and the compute performance of each. MQSim [46] is used to model flash performance. By combining one of the best simulators of the two worlds, the simulation results should be representative.

Gem5 is extended with scratchpad and streambuffer module [47] that features single cycle access to model AssasinSp, AssasinSb, and AssasinSb$ configurations. And the best-performant prefetcher, DCPTPrefetcher [43], in our benchmark is employed to evaluate Prefetch. At the same time of the simulation, access to scratchpads or streambuffers are traced to generate timed page-level IO traces which are further used as inputs to MQSim. Extended MQSim [48] simulates SSD internals and calculates the completion of each IO request, which we use to retime the computing process simulated by Gem5, i.e. adding additional latency if a page doesn't come out of the flash as early as compute engines modeled by Gem5 first access the specific page.

For UDP, UDPSim [42], the cycle-accurate simulator, is used instead of Gem5, but the SSD simulation and retiming by MQSim is the same.

Workload. We first evaluate ASSASIN with four standalone function offload to a computational SSD: Statistics, RAID4 erasure coding, RAID6 erasure coding, AES encryption. Each of these functions is in its own an application. And we program these functions in C++ through the programming model we discussed in Section V-D.

Then we consider TPC-H [49], which featured in much computational storage research [7], [8], [11], [13]–[15], [24], [36]. Our SparkSQL implementation [50] of TPC-H offloads Parse, Select and Filter operation through the datasource API [51] to computational storage simulated with Gem5 [25] or UDPSim [42]. Here we assume that the storage containing the TPC-H data would employ systematic coding (as in most erasure coded systems) so that source data are available even after coding, and that erasure coding blocks are large, reducing the boundary overhead (piecing together an object across SSDs/nodes).

B. *Single-function offload*

We evaluate the performance of different computational SSD configurations running Stat (summing a column), RAID4 and RAID6 erasure coding and AES encryption over the same 8 GiB data array serialized in binary flatly

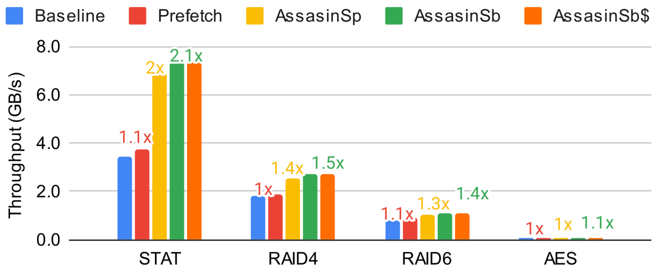


Figure 13: Throughput of offloaded standalone functions

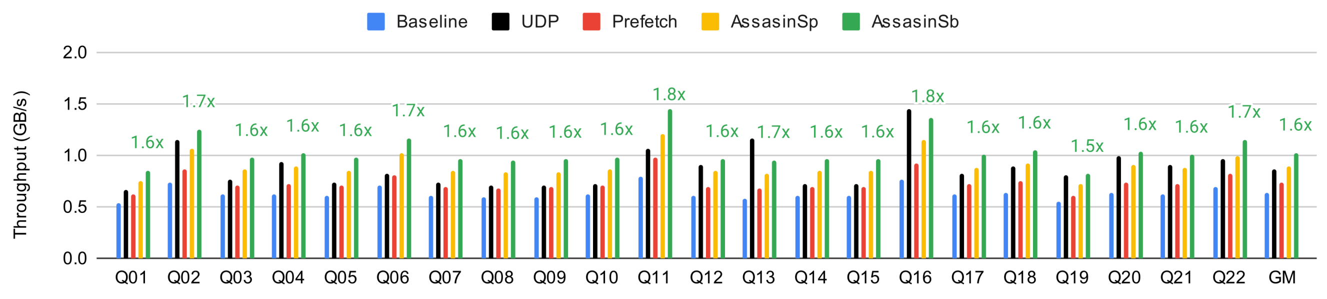


Figure 14: Computational SSD's throughput of offloaded PSF pipeline from TPC-H queries

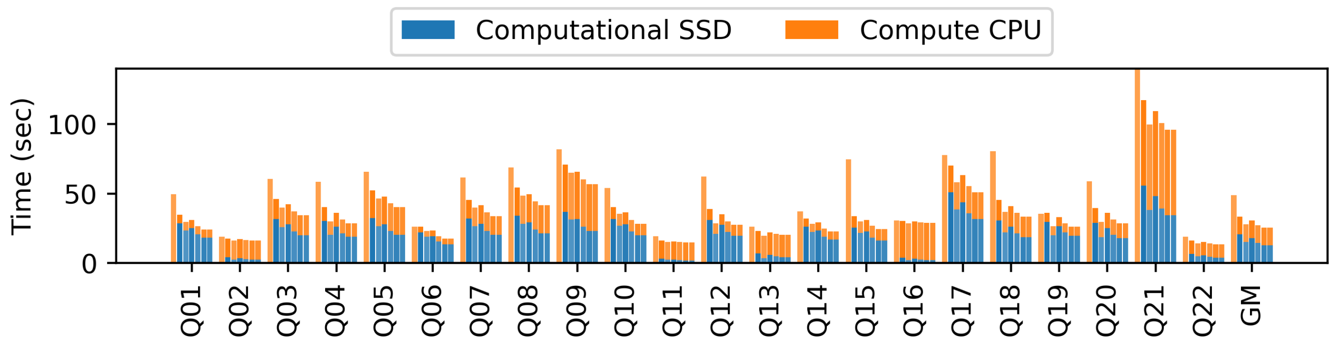


Figure 15: End-to-End latency. In each cluster: PureCPU, Baseline, UDP, Prefetch, AssasinSp, AssasinSb, AssasinSb$

(No deserialization or parsing is needed). Figure 13 shows the achieved throughput, with speedups over baseline labeled.

With the best prefetcher from Gem5, Prefetch is effective on reducing access latency. However, it only brings limited performance improvement because of the memory wall. The theoretically required DRAM bandwidth of Stat and RAID4 exceed what the LPDDR5 DRAM offers ($\sim$ 8 GB/s), leading to stalls for Prefetch. AssasinSp and AssasinSb address the memory wall issue through bypassing DRAM with Ping-Pong scratchpads or streambuffers, which leads to 1.3x-2.0x speedup for the first three functions and little to none memory bandwidth requirement. AssasinSb further outperforms AssasinSp by 10% on these cases with the stream ISA extension that enables automatic stream pointer management. AssasinSb and AssasinSb$ achieve the same performance as function states all fit in the scratchpad. Effectively the L1D cache provides no benefit (is not exercised by the program).

Please note that Stat, RAID4, RAID6 and AES are a sequence of functions with progressively greater computation intensity (ops per bytes). Generally, ASSASIN gives greater benefit for less compute-intensive functions, alleviating a memory bottleneck. As compute intensity increases, computing becomes the performance bottleneck. But it's worth pointing out that by employing acceleration, some originally compute-intensive functions (e.g. AES-NI for AES) could become memory-intensive and thus benefit from ASSASIN.

C. *Database function pipeline offload*

Now we look at computational SSD offload of a database function pipeline consisting of Parse, Select and Filter (PSF) from TPC-H workload with scaling factor 10 ($\sim$10GiB data), the system architecture of which is shown in Figure 12. Performance of offloaded functions is detailed in Figure 14.

UDP ISA [42] employs multiway dispatch and instructions that fuse operations to accelerate branch executions and unstructured data computation. This is the reason that UDP achieves on average (GeoMean) 1.3x speedup on offloaded PSF database function pipeline (this workload is well supported by UDP ISA specialization) over the Baseline.

On the other hand, we consider the progression of changes on Baseline's memory hierarchy while keeping general-purpose cores. PSF, bottlenecked by the Parse function, is moderate in terms of compute intensity. Prefetch achieves small speedup averaging (GeoMean) at 15% by hiding DRAM latency for accessing storage data. Adopting the idea of inline computation of storage data streams, AssasinSp matches UDP's speedup without UDP's exotic ISA customization. This comes from the low latency access to function states as well as storage data streams. Finally, both AssasinSb and AssasinSb$ further improve performance by 18% through the use of streambuffers and the stream ISA extension that eliminates the address calculations and pointer management instructions. This is 1.5x - 1.8x higher throughput than Baseline. The variation in speedup generally reflects the memory intensity of each offloaded pipeline.

Finally, we look into the overall data analytics performance

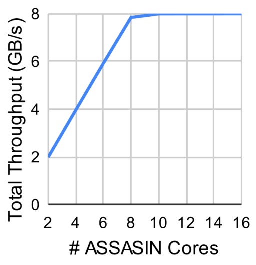


Figure 16: Compute performance scales linearly with cores.

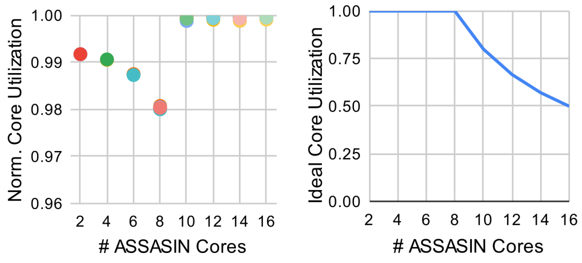


Figure 17: Normalized core utilization (left), normalized by ideal utilization (right).

for all 26 TPC-H queries, which stacks the host compute latency and computational SSD latency together. We also include the pure-CPU performance without computational SSD offload for comparison which essentially represents disaggregated storage architecture. Figure 15 details overall latency. Comparing to the Baseline, AssasinSb's 1.5x - 1.8x higher performance on in-storage compute translates to 1.1x - 1.5x end-to-end speedup which averages (GeoMean) at 1.3x. And please notice this is on top of the 1.9x speedup Baseline already brings over the no-computational-SSD pure-host-CPU (i.e. disaggregated storage) scenario.

D. Performance scalability

The ASSASIN SSD features an all-to-all interconnect between ASSASIN cores and flash channels (through flash controllers) as discussed in Section V-C to enable flexible performance scaling. Here we evaluate scalability and whether the crossbar interconnect potentially creates hot spots at a flash channel or causes ASSASIN cores to stall and wait for requested storage data. We consider a dummy workload where each ASSASIN core (AssasinSb variation) scans each byte of input, using the TPC-H datasets. If input data is always available, a 1 GHz core achieves 1 GB/s.

Various numbers of ASSASIN cores are considered to evaluate scalability. Figure 16 shows the achieved compute throughput of the computational SSD during scaling. Figure 17 shows the corresponding core utilization (normalized by the ideal utilization, derived by considering nominal bandwidth relationships between cores and channels). As shown, the interconnect allows linear scaling of compute performance until bounded by flash array throughput (8GB/s in total). Cores have high utilization, more than 98%, reflecting that the flash array and the interconnect deliver pages in time to keep the cores busy. Further, flash channels also feature balanced high throughput as shown in Figure 18. This is because independent FTL (FTL modeled by MQSim is employed here) already aims to distribute pages evenly across the channels for better storage performance, separate from computational SSD considerations.

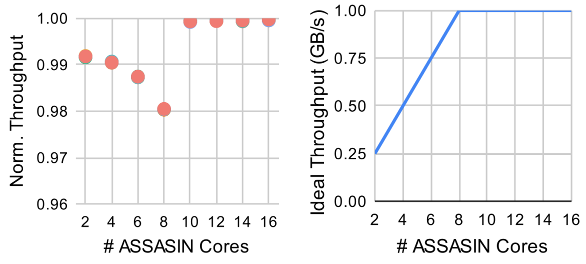


Figure 18: Ext4/MQSim layout combined with Xbar produces even load across flash channels

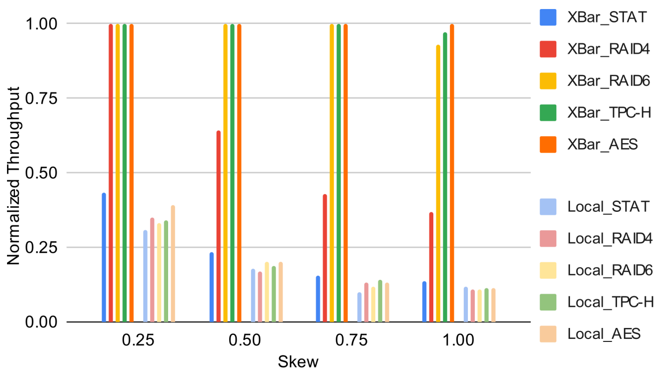


Figure 19: Performance sensitivity to skew across channels (flash data layout)

E. Sensitivity to flash array data layout skew

We further evaluate ASSASIN's (AssasinSb variation) performance sensitivity to flash array data layout skew, which is defined as (D_i denotes the amount of the to-be-processed data in the i-th channel):

$$Skew = \frac{1}{n-1} max_i(\frac{D_i}{avg_i(D_i)}) \qquad Skew \in [0,1]$$

ASSASIN with the SSD-level crossbar interconnect to redistribute flash data to compute engines is compared with the alternative architecture (Figure 7) from application-specific computational storage [30], [37], [39]. Ignoring FTL generality issues of the alternative architecture discussed in Figure 7 and Section V-A, ASSASIN cores are switched in at each channel to perform channel-local compute. Four layouts with varied skew for requested data are evaluated (from low skew (Skew=0.25) to extreme skew (Skew=1)) in additional to the no skew

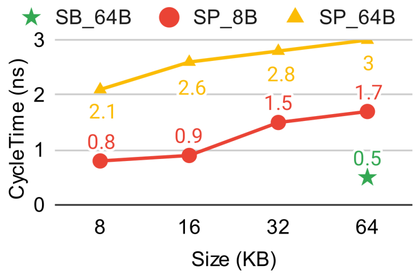


Figure 20: Timing for ASSASIN MemArch extensions (SB = Streambuffer, SP=Scratchpad)

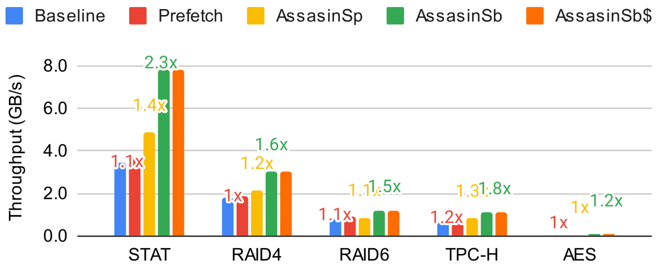


Figure 21: Throughput after timing adjustment (TPC-H is GeoMean across queries))

senario. Respective performance normalized to no skew one are shown in Figure 19.

The two clusters of bars show that the crossbar architecture consistently outperforms the channel-local compute architecture in the presence of skew. Further as skew increases the benefits increase to as much as 8-fold.

Within the XBar cases, for flash-read limited functions like Stat, uneven layout aggravates the storage read bottleneck, and XBar-based global compute helps less. But for compute-limited functions like RAID6, TPC-H and AES, XBar-based global compute can source pooled compute engines effectively for the data read out from the most-skewed channel, matching the overall throughput with that of the even data layout scenario.

We conclude that ASSASIN's XBar interconnect architecture achieves robust performance, thereby enabling flexible, independent FTL layout mananagement. Benefits compared to channel-local proposals (as in Figure 7) can be as large as 3-8x.

F. Clock-speed benefits and adjusted performance

To assess the clock-speed benefits of ASSASIN's streaming architecture, we implement the ISA extension described in Section V-B, using SystemVerilog and with SAED14nm [52]. The implemented streambuffer (for AssasinSb) provides 1B-64B access to the stream heads in the 'MEM' pipeline stage of each ASSASIN core. As a comparison, scratchpads (for AssasinSp) with varied widths (8B for a scalar core, 64B for SIMD extensions) and sizes are implemented.

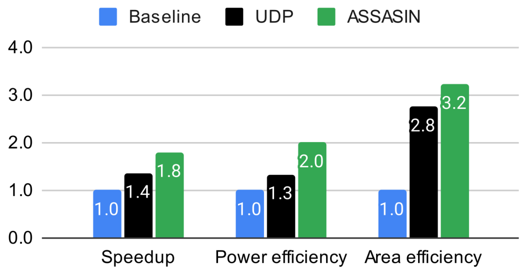


Figure 22: Relative speedup, relative power efficiency and relative area efficiency

Our results (see Figure 20) show AssasinSb's streambuffer achieves 0.5ns per cycle even with a wide, i.e. 64B, interface provisioned for SIMD. A small and prefetched FIFO, built upon restricted streaming semantics (head-only) of 'StreamLoad' and 'StreamStore' instructions, on top of streambuffer with coarse-grained (128B aligned) accesses is the key to this high speed.

In contrast, the scratchpad implementations are significantly slower because they require random access (large MUXes / access trees). At 64KB, the scratchpad even with a narrow 8B interface, requires 2 cycles for each access in a 1GHz core. Thus, AssasinSp where most accesses are served by the scratchpads would take performance penalty.

We consider these results in the context of a RISC-V core, with a classical five-stage pipeline (IF, DE/RR, EX, MEM, WB). In this design, the prefetch FIFO for the streambuffer is added where the dcache access would occur. Substituting dcache to much faster streambuffer allows the clock period to be reduced by 11% (the critical path shifts to 'IF'). On the other hand, scratchpads would have to be timed for 2-cycle accesses, and without any cycle time benefits.

Finally, we adjust the performance based on above findings, as shown in Figure 21. Overall, AssasinSb's throughput improves to 1.5x-2.4x (from 1.4-2.1x) over Baseline, resulting from the cycle time reduction. AssasinSp, with one additional cycle needed for scratchpads' accesses, degrades to only 1.1x-1.4 (from 1.3 to 2x). In short, on top of the DRAM bypassing feature, AssasinSb's streaming memory architecture and instructions delivers a further 1.5x increase in performance.

G. Power and area efficiency

We evaluate the cost of the in-SSD computing engines and their memory hierarchy in terms of power and silicon area of different configurations. Synopsys design compiler and 14nm SAED technology library [52] are used to evaluate the costs of logic. The in-order RISC-V cores we use in baseline and ASSASIN (i.e. AssasinSb variation) are based on ibex [44]. For UDP, we evaluate with its SystemVerilog-based ASIC implementation. And Cacti [26] is used for caches, streambuffers and scratchpads.

Table V: Power and silicon area

	Ibex core	UDP core	32KB 8way $	256KB 16way $	64KB SRAM	Crossbar	**Baseline**	**UDP**	**ASSASIN**
Power(mw)	0.241	1.210	0.379	1.136	0.145	0.439	14.048	14.333	12.625
Area(mm^2)	0.039	0.059	0.044	0.282	0.030	0.005	2.923	1.444	1.625

The power and silicon area costs for subcomponents and three evaluated configurations are summarized in Table V. One thing worth pointing out is that a L1 cache or similar-size SRAM are at the same order of magnitude with the compute logic of a core in area and power. This shows the significance of memory hierarchy innovation as pursued by ASSASIN. The speedup over baseline, relative power efficiency (speedup per unit power) and relative area efficiency (speedup per unit area) of each configuration are plotted in Figure 22. ASSASIN (i.e. AssasinSb) achieves 2.0x and 3.2x higher power and area efficiency comparing to Baseline through its streambuffer and scratchpad based memory hierarchy and streaming instruction-set extension. And ASSASIN(i.e. AssasinSb) with general-purpose RISCV cores also outperforms a UDP accelerator which employs an exotic customized ISA for unstructured data processing.

VII. Related work

Pioneering work includes "Active Storage" in the 1990s [18], [19] that focused on rotating hard disks in a single system, not the modern context of shared cloud storage services with high-performance flash SSDs. More recent efforts study computational storage in SSD systems. Some propose general [11], [22]–[24] or application-specific (i.e. data analytics) [8], [13], [15] software architecture on top of general-purpose hardware architectures as shown in Figure 4. Others propose application-specific hardware and associated software architectures for deduplication [28], [29], key-value storage [10], [14], deep learning [30], [37], graph analytics [31], [32], and data analytics [7], [9], [34] in computational SSDs. Such studies showcase the benefits of offloading (comparing to non-offloading) or application-specific hardware customization, but give little insight as what computation structures/properties enables system efficiency when employing computational SSD and how to build general-purpose hardware architecture support (with associated software architecture adaptations) based on these properties, which is the essence of our ASSASIN study.

General-purpose hardware architecture for computational SSD. QuerySSD [35] pioneered the database-oriented function offloads and assessed the speedups and energy benefits. ActiveFlash [15] showcases similar computational SSD potential but for data reduction functions from scientific computing workloads. Biscuit [13] advances the art with a flow-based programming model for computational SSD offload. Summarizer [11] materializes the system software architecture design including a detailed NVMe command interface. YourSQL [8] provides richer software operator support to enable the offload of all TPC-H queries. BlockIF [23] argues for a software architecture for computational storage that conforms to traditional block-oriented storage interface for increased adoption. IceClave [20] proposes a low-overhead trusted execution environment for in-SSD computations and advances on the security front.

These software and system architecture advancement are all based on the general-purpose hardware architecture for computational SSDs where compute engines are embedded-class general purpose cores computing on top of the conventional cache-DRAM memory hierarchy, as depicted in Figure 4, and thus amenable to the in-SSD memory wall. ASSASIN builds on top of these software architecture innovations in terms of NVMe command adaption and block-interface conformance but advances the hardware architecture with efficient flash data stream handling and addresses the in-SSD memory wall.

Application-specific hardware architectures in computational SSD. DedupInSSD [29] proposes hardware hash acceleration for in-SSD deduplication and showcase the SSD write latency reduction and lifespan improvement. CIDR [28] proposes more-tailed scalable hardware architecture for deduplication which utilizes additional scratchpads for signature management and integrates compress engines. LightStore [10] augments the general purpose architecture with a hardware network module to expose a scalable key-value storage onto the datacenter network. Caribou [14] further maps the flash management into hardware modules of cuckoo hash and slab allocation manager and also supports hardware accelerated in-SSD processing primitives like select and compression. GList [31] proposes to integrate hardware sampling unit and a PE array in the SSD board for graph learning workloads to enjoy internal excessive storage bandwidth. GrafBoost [32] employs a sort-reduce accelerator in storage for vertex-centric graph processing. Deepstore [30] integrates systolic arrays at SSD, channel and chip-level for accelerated neural network inference. Thrifty [37] proposes chip-level binary compute accelerator and SSD-level training accelerator for hyper-dimension compute training.

Despite employing application-specific hardware architecture, in all above work except DeepStore and Thrifty, compute engines source flash data via SSD DRAM, thus amenable to SSD DRAM bottleneck. And the employment of hardware acceleration modules actually further increases the bandwidth requirements of SSD DRAM, as discussed in Section VI-B. Although channel and chip-level compute engine integration employed by Deepstore and Thrifty has the potential of addressing the SSD memory wall, it further requires application-specific control on data (SSD page) layout for parallelism exploitation. This is fine for neural network applications featuring regularly-shaped tensors which can be easily split into the flash array, but a deal breaker for general workloads with diverse basic unit sizes. ASSASIN, through pooling compute elements at the SSD-level, supports different basic unit sizes by piecing a unit form pages across channels and dies. ASSASIN addresses the memory wall while leaving the flash translation layer and the SSD interface unchanged.

FPGA-based hardware architecture for computational SSD. There is a middle ground of computational SSDs where an FPGA chip is integrated into the SSD board and assumes the computation responsibility [53]. This allows the storage vendor to either support more flexible and fast iterations through hardware reconfiguration or shift the responsibility of architecting the compute engines including both general-purpose or application-specific ones in computational SSDs to the customer, but at the cost of both increased power and silicon area compared with a fixed hardware architecture, and ease of use for requiring hardware expertises for reconfiguring the FPGA.

Both Insider [24] and Access [22] try to address the ease-of-use aspects by providing system software architecture support of file system filter [24] or pipe [22] abstractions for the FPGA computing kernels to enable easy system pipelines and embracing high-level synthesis for developing these kernels. However, because the SSD FPGA still fetches the data from SSD DRAM or even requires data to be staged in an additional FPGA DRAM [53], the in-SSD memory wall is left unaddressed. The workload and architecture knowledge distilled in ASSASIN is transferable to the FPGA-based architectures for computational SSDs.

Disaggregated Storage. Disaggregated storage is a different approach than computational storage. It has the advantage of separate management and scaling of compute and storage, thus allowing compute to scale out and catch up with continuously improving flash bandwidth. Good examples of disaggregated storage include JBOF with NVMe-oF [54] and Amazon EBS [55]. However, comparing to computational storage, it doesn't offer the ability of matching more effective storage bandwidth to a single compute unit (a single point of consistency), and requires higher interconnect bandwidths and also puts high burdens of packets and data processing on the compute processors.

A related trend is Open-channel SSD [56] and Zoned Namespace SSD [57] that enable better control of placement and performance by externalizing management. But they do not address the system architecture aspects (driving more storage bandwidths, reducing interconnect bandwidth requirement, offloading packet and data processing) which computational storage strives for.

VIII. Summary and Future Work

The memory wall problem in the computational storage emerges when compute in storage tries to match the flash bandwidth. To characterize the problem clearly, we studied a broad range of research proposals for domain-specific properties that allow architecture innovations. Key insight arises that computational storage functions all feature implementations with streaming accesses to storage data and random accesses to function states of limited size.

Inspired by this insight, we designed ASSASIN, which allows inline stream computing on flash data streams through a hybrid high-bandwidth memory hierarchy composed of stream buffers and scratchpad and a streaming instruction-set extension. ASSASIN achieves 1.5x - 2.4x speedup on functions offloaded in storage which translates to 1.1x - 1.5x overall end-to-end speedup, compared to the state-of-the-art general-purpose computational SSD architecture.

There are a number of interesting directions for future work. Further distilling workload properties to shape an efficient compute ISA extension for computational SSDs is intriguing, as ASSASIN addresses the memory-intensive part. Since streaming workloads are also prevalent in general-purpose computing (i.e. servers in the cloud) environment, 'backporting' streaming architecture extension for computational storage to 'upstream' is also interesting.

Acknowledgment

We thank reviewers for insights that improved the paper significantly. Insights on SSD's internal architecture from Hui Zhang and Pratik Mishra (then of Samsung MSL) are also much appreciated. This work is supported by a grant from Samsung MSL, NSF Award CNS-1909364, and the CERES Center for Unstoppable Computing.

References

[1] "Samsung Sixth-Generation V-NAND SSDs," Samsung News Room.

[2] "Samsung PM9A3 Data Center SSD," https://www.samsung.com/semiconductor/ssd/pm9a3/.

[3] "Samsung 980 Pro SSD," https://s3.ap-northeast-2.amazonaws.com/global.semi.static/Samsung-NVMe-SSD-980-PRO-Data-Sheet_Rev.2.1.pdf.

[4] 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.

[5] Y. Kang, R. Pitchumani, P. Mishra, Y.-s. Kee, F. Londono, S. Oh, J. Lee, and D. D. Lee, "Towards building a high-performance, scale-in key-value storage system," in *Proceedings of the 12th ACM International Conference on Systems and Storage*. ACM, 2019, pp. 144–154.

[6] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon aurora: Design considerations for high throughput cloud-native relational databases," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1041–1052.

[7] L. Woods, Z. István, and G. Alonso, "Ibex: an intelligent storage engine with support for advanced sql offloading," *Proceedings of the VLDB Endowment*, vol. 7, no. 11, pp. 963–974, 2014.

[8] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. Lee, and J. Jeong, "Yoursql: a high-performance database system leveraging in-storage computing," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 924–935, 2016.

[9] C. Zou, H. Zhang, A. A. Chien, and Y.-S. Ki, "Psacs: Highly-parallel shuffle accelerator on computational storage." in *ICCD*, 2021, pp. 480–487.

[10] C. Chung, J. Koo, J. Im, S. Lee *et al.*, "Lightstore: Software-defined network-attached key-value drives," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 939–953.

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

[12] C. Zou, A. A. Chien, R. Gardner, and I. Vukotic, "Computational storage to increase the analysis capability of tier-2 hep data sites," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 803–804.

[13] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho *et al.*, "Biscuit: A framework for near-data processing of big data workloads," in *ACM SIGARCH Computer Architecture News*, vol. 44. IEEE Press, 2016, pp. 153–165.

[14] Z. István, D. Sidler, and G. Alonso, "Caribou: intelligent distributed storage," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1202–1213, 2017.

[15] D. Tiwari, S. Boboila, S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin, "Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013, pp. 119–132.

[16] H. Li, M. Hao, S. Novakovic, V. Gogte, S. Govindan, D. R. Ports, I. Zhang, R. Bianchini, H. S. Gunawi, and A. Badam, "Leapio: Efficient and portable virtual nvme storage on arm socs," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 591–605.

[17] "What is computational storage," https://www.snia.org/education/what-is-computational-storage.

[18] E. Riedel, G. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia applications," in *Proceedings of 24th Conference on Very Large Databases*. ACM, 1998, pp. 62–73.

[19] A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VIII. ACM, 1998, pp. 81–91.

[20] L. Kang, Y. Xue, W. Jia, X. Wang, J. Kim, C. Youn, M. J. Kang, H. J. Lim, B. Jacob, and J. Huang, "Iceclave: A trusted execution environment for in-storage computing," in *Proceedings of the 54th International Symposium on Microarchitecture*. ACM, 2021.

[21] "ONFI Specifications," https://www.onfi.org/specifications.

[22] 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.

[23] I. F. Adams, J. Keys, and M. P. Mesnier, "Respecting the block interface–computational storage using virtual objects," in *11th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[24] 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.

[25] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[26] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[27] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 15–26.

[28] M. Ajdari, P. Park, J. Kim, D. Kwon, and J. Kim, "Cidr: A cost-effective in-line data reduction system for terabit-per-second scale ssd arrays," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 28–41.

[29] J. Kim, C. Lee, S. Lee, I. Son, J. Choi, S. Yoon, H.-u. Lee, S. Kang, Y. Won, and J. Cha, "Deduplication in ssds: Model and quantitative analysis," in *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2012, pp. 1–12.

[30] V. S. Mailthody, Z. Qureshi, W. Liang, Z. Feng, S. G. d. Gonzalo, Y. Li, H. Franke, J. Xiong, J. Huang, and W.-m. Hwu, "Deepstore: In-storage acceleration for intelligent queries," in *Proceedings of the 52th International Symposium on Microarchitecture*. ACM, 2019.

[31] C. Li, Y. Wang, C. Liu, S. Liang, H. Li, and X. Li, "{GLIST}: Towards {In-Storage} graph learning," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 225–238.

[32] S.-W. Jun, A. Wright, S. Zhang, S. Xu *et al.*, "Grafboost: Using accelerated flash storage for external graph analytics," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 411–424.

[33] D. R. Horn, K. Elkabany, C. Lesniewski-Lass, and K. Winstein, "The design, implementation, and deployment of a system to transparently compress hundreds of petabytes of image files for a file-storage service," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 1–15.

[34] S. Kang, J. An, J. Kim, and S.-W. Jun, "Mithrilog: Near-storage accelerator for high-performance log analytics," in *Proceedings of the 54th International Symposium on Microarchitecture*. ACM, 2021.

[35] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart ssds: Opportunities and challenges," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 1221–1230.

[36] J. LeFevre and N. Watkins, "Skyhook: Programmable storage for databases." Boston, MA: USENIX Association, Feb. 2019.

[37] S. Gupta, J. Morris, M. Imani, R. Ramkumar, J. Yu, A. Tiwari, B. Aksanli, and T. Š. Rosing, "Thrifty: Training with hyperdimensional computing across flash hierarchy," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9.

[38] K. K. Matam, G. Koo, H. Zha, H.-W. Tseng, and M. Annavaram, "Graphssd: graph semantics aware ssd," in *Proceedings of the 46th international symposium on computer architecture*, 2019, pp. 116–128.

[39] N. Mansouri Ghiasi, J. Park, H. Mustafa, J. Kim, A. Olgun, A. Gollwitzer, D. Senol Cali, C. Firtina, H. Mao, N. Almadhoun Alserr *et al.*, "Genstore: a high-performance in-storage processing system for genome sequence analysis," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 635–654.

[40] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: Base user-level isa," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, vol. 116, 2011.

[42] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien, "Udp: a programmable accelerator for extract-transform-load workloads and more," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017.

[41] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun *et al.*, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 654–667.

[43] M. Grannaes, M. Jahre, and L. Natvig, "Storage efficient hardware prefetching using delta-correlating prediction tables," *Journal of Instruction-Level Parallelism*, vol. 13, pp. 1–16, 2011.

[44] P. D. Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE, 2017, pp. 1–8. [Online]. Available: https://github.com/lowRISC/ibex

[45] "RISC-V GNU Compiler Toolchain," https://github.com/riscv/riscv-gnu-toolchain.

[46] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "Mqsim: A framework for enabling realistic studies of modern multi-queue {SSD} devices," in *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, 2018, pp. 49–66.

[47] "Extended Gem5 for ASSASIN evaluations," https://github.com/compstorassasin/gem5.

[48] "Extended MQSim for ASSASIN evaluations," https://github.com/compstorassasin/MQSim.

[49] "TPC-H dataset," http://www.tpc.org/tpch/.

[50] "SparkSQL TPC-H implementaion with Computational Storage Offload," https://github.com/compstorassasin/compstor.

[51] "Introducing Apache Spark Data Sources API V2," https://databricks.com/session/apache-spark-data-source-v2.

[52] "Synopsys teaching resources," https://www.synopsys.com/community/university-program/teaching-resources.html.

[53] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart ssds: opportunities and challenges," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 1221–1230.

[54] "NVMe-OF JBOF," https://nvmexpress.org/wp-content/uploads/NVMe-202-1-Part-1-JBOFs_Final.pdf.

[55] "Amazon EBS: SSD," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html.

[56] M. Bjørling, J. Gonzalez, and P. Bonnet, "Lightnvm: The linux open-channel {SSD} subsystem," in *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, 2017, pp. 359–374.

[57] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. Moal, G. Ganger, and G. Amvrosiadis, "Zns: Avoiding the block interface tax for flash-based ssds," in *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*, 2021.