

OPENCXD: An Open Real-Device-Guided Hybrid Evaluation Framework for CXL-SSDs

Hyunsun Chung^{1,*}, Junhyeok Park^{1,*}, Taewan Noh¹, Seonghoon Ahn¹

Kihwan Kim¹, Ming Zhao², Youngjae Kim^{1,†}

¹Sogang University, Seoul, Republic of Korea, ²Arizona State University, Tempe, AZ, USA

Abstract—The advent of Compute Express Link (CXL) enables SSDs to participate in the memory hierarchy as large-capacity, byte-addressable memory devices. These CXL-enabled SSDs (CXL-SSDs) offer a promising new tier between DRAM and traditional storage, combining NAND flash density with memory-like access semantics. However, evaluating the performance of CXL-SSDs remains difficult due to the lack of hardware that natively supports the CXL.mem protocol on SSDs. As a result, most prior work relies on hybrid simulators combining CPU models augmented with CXL.mem semantics and SSD simulators that approximate internal flash behaviors. While effective for early-stage exploration, this approach cannot faithfully model firmware-level interactions and low-level storage dynamics critical to CXL-SSD performance. In this paper, we present OPENCXD, a real-device-guided hybrid evaluation framework that bridges the gap between simulation and hardware. OPENCXD integrates a cycle-accurate CXL.mem simulator on the host side with a physical OpenSSD platform running real firmware. This enables in-situ firmware execution triggered by simulated memory requests. Through these contributions, OPENCXD reflects device-level phenomena unobservable in simulation-only setups, providing critical insights for future firmware design tailored to CXL-SSDs.

Index Terms—Compute Express Link, Solid-State Drive

I. INTRODUCTION

The growing scale of modern deep learning models and data analytics applications has led to memory footprints reaching tens of terabytes [1]–[3], far beyond the limits of traditional DRAM installations. This widening gap between demand and capacity has resurrected the infamous *memory wall* [4], in which memory bandwidth and size become critical bottlenecks to system performance. To mitigate this issue, researchers have begun exploring memory expansion techniques that repurpose alternative technologies as additional memory [5]–[7], paving the way for new architectural paradigms. Among these, Compute Express Link (CXL) [8] has emerged as a promising enabler of such memory expansion. CXL is a high-bandwidth, cache-coherent interconnect built on top of the PCI Express (PCIe) [9] infrastructure. By using CXL, large-capacity PCIe devices like flash-based Solid-State Drives (SSDs) can be attached directly to the host system memory space, creating a memory pool that augments or disaggregates DRAM [10].

Notably, CXL’s memory semantics (CXL.mem) support byte-addressable access to device memory, meaning a CPU can read or write an SSD’s onboard DRAM buffer with ordinary load and store instructions. This eliminates the need for traditional I/O commands, significantly reducing software

overhead and access latency while enabling a unified, tiered memory architecture that extends beyond DRAM’s capacity limits [11]. Building on this capability, a new class of memory-semantic devices, CXL-enabled SSDs (CXL-SSDs), has begun to emerge (§II). These devices leverage mature NAND flash technology to offer terabytes of byte-addressable capacity at a fraction of DRAM’s cost per gigabyte [7], while maintaining access latencies in the microsecond range. Although slower than DRAM by several orders of magnitude, CXL-SSDs provide significantly lower latency than traditional SSDs accessed via the block interface (e.g., NVMe [12]).

The key challenge for CXL-SSDs is how to architect and evaluate these devices effectively, maximizing their performance potential while addressing inherent latency trade-offs. However, this evaluation remains difficult in practice, due to *the lack of hardware that natively supports the CXL.mem protocol on SSDs*. To overcome this, recent research has adopted software-based hybrid simulators, combining CPU simulators (e.g., MacSim [13], Gem5 [14]) extended with CXL.mem semantics and SSD simulators (e.g., SimpleSSD [15], FlashSim [16]) that model internal flash behaviors such as address translation and I/O scheduling. This methodology enables early-stage design exploration and has been widely used in prior work [17]–[19].

Notably, the CXL.mem interface overhead itself has been characterized in prior studies [20], [21], and shown to be relatively consistent and bounded [10]. As such, injecting this CXL interface time overhead into x86 simulations as a parameter is generally considered a reasonable approach for modeling host-side access costs. However, *the same cannot be said for modeling device-side behavior*. Replacing a real SSD with a simulator introduces significant challenges in capturing the full complexity of CXL-SSD-specific firmware logic and storage interactions, which are limitations that fundamentally hinder accurate evaluation of CXL-SSD designs (§III).

There are two key limitations of simulation-only evaluation. **First**, since CXL-SSDs function as memory rather than storage, they must handle fine-grained, cacheline-level memory accesses, making them highly sensitive to device-side performance fluctuations. However, simulators typically rely on static latency models, overlooking dynamic behaviors such as real-time NAND latency variability and firmware delays, resulting in latency estimation errors as high as 36% [22]. **Second**, CXL-SSDs introduce new firmware-managed mechanisms, such as write logging and log compaction [11], that do not exist in conventional SSDs. Simulating these new

*They are first co-authors and have contributed equally.

†Y. Kim is the corresponding author.

behaviors solely from the host side, without executing them on real hardware and capturing internal device interactions, fails to reflect crucial contention and scheduling effects, yielding an incomplete and often misleading performance picture [23].

In this paper, we present OPENCXD, a real-device-guided hybrid evaluation framework for CXL-SSDs. OPENCXD combines a cycle-accurate CPU and memory simulator augmented with CXL.mem support and a hardware platform based on OpenSSD [23], an open-source SSD prototype that runs actual controller firmware. In essence, the host side of the system (CPU, cache, and memory controllers) is simulated, allowing full control over experimental scenarios and observability of system-level events, while the storage side is handled by a physical device that embodies a CXL-SSD (§IV).

We address two key challenges in developing OPENCXD:

- **Enabling Cacheline-Level Access over NVMe.** Unlike DRAM or DRAM-based CXL memory modules [24], publicly available SSD prototypes do not natively support cacheline-sized CXL.mem transactions. Bridging this gap is non-trivial because NVMe operates on fixed-sized (e.g., 4 KB) blocks with DMA-based transfers, not on byte-addressable memory [25], [26]. To address this, OPENCXD defines custom NVMe commands that encode CXL.mem semantics and emulate cacheline-granularity memory access without modifying the physical hardware interface.
- **Cycle-Level Timing Integration with Real Firmware.** While the cycle-accurate host simulator tracks timing at the cycle level, the OpenSSD operates asynchronously with real-time firmware execution. OPENCXD resolves this by adopting a *device-in-the-loop* design: for each memory access, the simulator pauses execution, delegates the operation to OpenSSD running real firmware, and waits until the firmware measures the end-to-end latency and reports it to the host simulator. The simulator then resumes by converting the measured latency to cycles using a calibrated timing ratio and advances its internal clock accordingly.

We then implement key firmware components of a state-of-the-art CXL-SSD [11] within the OpenSSD device. This allows fine-grained memory requests from the host to trigger CXL-SSD firmware execution. These design choices enable OPENCXD to maintain the configurability and observability of full-system simulation, while introducing *hardware realism* through in-situ execution of the SSD’s software stack.

Our evaluation shows that OPENCXD captures performance characteristics from real-life hardware unobservable in software SSD simulations. These include $2.4\times$ higher NAND read latencies due to lower-level NAND controller and SSD firmware overheads, as well as DRAM latency spikes over $2\mu s$. Such evaluation highlight the need for real-device-guided evaluation to uncover nuanced CXL-SSD dynamics such as out-of-order persistence and timing variability.

This work makes the following contributions:

- **Real-Device-Guided CXL-SSD Evaluation Framework:** We present OPENCXD, the first hybrid platform that combines a full-system CXL.mem simulator with an open-source SSD prototype running real firmware, enabling accurate and practical evaluation of CXL-SSD architectures.

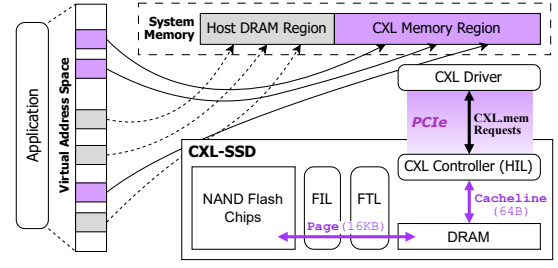


Fig. 1: System architecture of CXL-SSD.

- **Device-in-the-Loop Architecture:** We develop a tightly coupled host-device interface that allows simulated memory accesses to trigger firmware execution and measure and integrate its latency into host-side simulation in situ.
- **Implementation of CXL-SSD Internals on Hardware:** We implement key CXL-SSD components, including write log and log index, on actual hardware, revealing behaviors and bottlenecks that prior simulation-only setups miss.

II. BACKGROUND

A. Hardware Architecture of CXL-SSDs

The hardware architecture of CXL-SSDs is composed of the same core components as conventional SSDs [7], [11], [27], including NAND, DRAM, and a System-on-Chip (SoC) controller. CXL-SSDs differ from traditional SSDs in how the device is integrated into the system: they function as CXL.mem endpoints, exposing their capacity as part of the host’s memory address space. When the CPU accesses a CXL memory region, cacheline-sized memory requests (typically 64 B) are issued via the CXL driver, and the Host Interface Layer (HIL) of the SSD controller buffer them in the on-board DRAM, and eventually flush them to NAND in page-sized units (e.g., 16 KB) through the Flash Translation Layer (FTL) and Flash Interface Layer (FIL), as illustrated in Fig. 1.

B. Software Architecture of CXL-SSDs

SkyByte [11] represents the state-of-the-art design of a CXL-SSD and provides a detailed explanation of its internal software architecture, as illustrated in Fig. 2. The key software components inside the CXL-SSD include a *Write Log*, *Data Cache*, and *Log Index*, which bridge the granularity mismatch between 64 B cacheline accesses and NAND page-level operations. The *Write Log* buffers incoming CXL.mem write requests at 64 B granularity. The *Data Cache* functions similarly to a traditional SSD’s in-memory NAND page cache, managing recently accessed NAND pages. Additionally, to mitigate the latency penalty from NAND I/O, SkyByte incorporates context switching to perform other I/O requests while said NAND I/O operation is ongoing.

Fig. 2(a) illustrates the write path. When a CXL.mem write arrives, ① the device first stores the cacheline-sized payload into the *Write Log*. ② If the corresponding NAND page is resident in the *Data Cache*, the cacheline update is also applied there to maintain consistency. ③ The system then updates the *Log Index* to track the location of the buffered write.

Fig. 2(b) illustrates the read path. When a CXL.mem read arrives, ① the system checks whether the target cacheline is

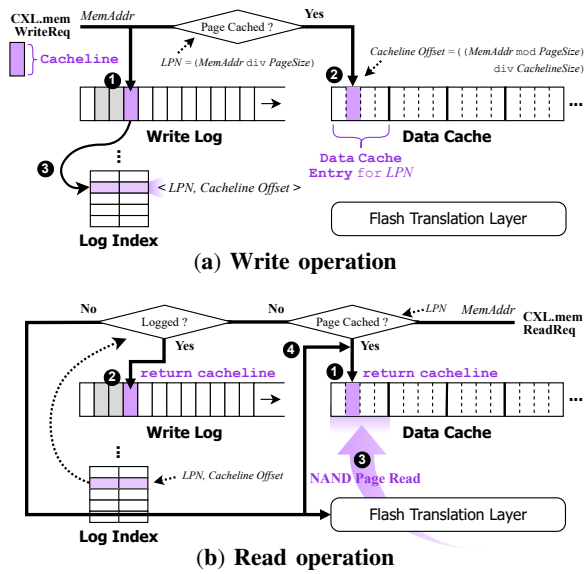


Fig. 2: Write/read flows of the state-of-the-art CXL-SSD [11], comprising a *Write Log*, *Data Cache*, and *Log Index*.

present in the *Data Cache*; if so, it is returned directly to the host. ② If the *Data Cache* does not contain the cacheline but the *Write Log* does, the system retrieves and returns the buffered version. In cases where the cacheline is found in neither buffer, ③ the corresponding NAND page is read into the *Data Cache*, and ④ the cacheline is served from there. During this page load, existing NAND pages in the *Data Cache* may be evicted and flushed back to NAND.

To manage and track buffered writes, SkyByte employs a two-level *Log Index*: the first level identifies modified NAND pages, while the second level maps individual cacheline offsets within those pages. To reclaim log space and persist updates, the system periodically performs **log compaction**. During compaction, the system first scans the first-level index to identify NAND pages that contain valid log entries. If such a page is already present in the *Data Cache*, it is flushed directly to NAND. Otherwise, the NAND page is loaded into memory. It then consults the second-level index to locate all valid, buffered cachelines associated with the page and merges them into the in-memory copy. Finally, the merged page is written back to NAND, and the log entries are invalidated.

III. MOTIVATION

A. Limitations of Purely Software-Driven SSD Simulation

Limitation #1) Reliance on Parameter-Driven Calculation: Accurate characterization of NAND I/O latency is critical for optimizing CXL-SSDs, where all memory accesses operate at cacheline granularity and DRAM cache misses directly translate to NAND operations (§II-B). As CXL-SSDs function as memory-semantic devices, they are highly sensitive to the latency path between on-board DRAM and NAND. Therefore, understanding and optimizing these paths require precise visibility into device-level behavior.

However, most SSD simulators, including SimpleSSD [15] used in SkyByte, rely on parameter-driven, static latency

modeling to estimate NAND flash behavior [28]. These models use predefined latency values of the NAND flash used in the SSD as input parameters to derive NAND read and program latency [29], ignoring dynamic factors such as flash controller scheduling and internal firmware optimizations. While efforts have been made to address this issue, for example, SimpleSSD now models ARM cores and flash parallelism (channels, ways, dies, planes) to simulate NAND I/O scheduling, these remain abstractions built atop assumed hardware parameters. Indeed, SimpleSSD reports deviations of up to 28% in throughput and 36% in latency compared to real hardware [22]. This limitation is also evident in our experiments (§III-B), where static latency modeling fails to reflect real-world NAND I/O behavior.

Limitation #2) Missing Hardware-Grounded Firmware Validation:

A side effect of using software-driven SSD simulations is as no real SSD hardware is involved, all code related to SSD firmware is run on a host system. This environment can lead to inaccurate assumptions of what is possible in SSD firmware for implementing CXL-SSD optimizations. SkyByte works around this limitation by implementing their optimizations, such as the *Write Log*, *Log Index*, and *Data Cache*, in prototype FPGA hardware, and use the averages of operation measurements in the simulation [11]. While this approach enables partial evaluation grounded in real hardware, it comes short of a holistic assessment across the full CXL-SSD stack. Specifically, it cannot capture the dynamic interaction among DRAM, NAND flash, and ARM cores triggered by real-time CXL.mem requests during workload execution (§V).

The approach also faces previously mentioned pitfalls of simulations using statically declared parameters for operational overheads in CXL-SSD evaluation. Although certain firmware components are executed on prototype hardware, the measured latencies are averaged and reused as static inputs in the simulator, making the evaluation effectively parameter-driven. As a result, it cannot capture the dynamic, request-level interactions that emerge from executing the full firmware stack on real SSD hardware under live CXL.mem workloads.

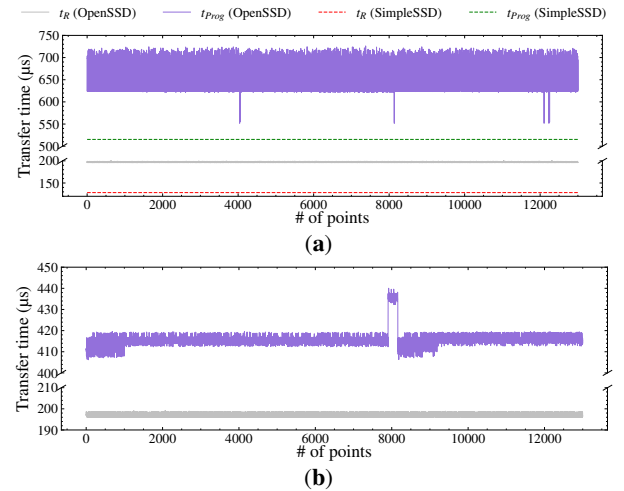


Fig. 3: NAND read/program I/O times of two different types of NAND (a) and (b) with $idepth=1$.

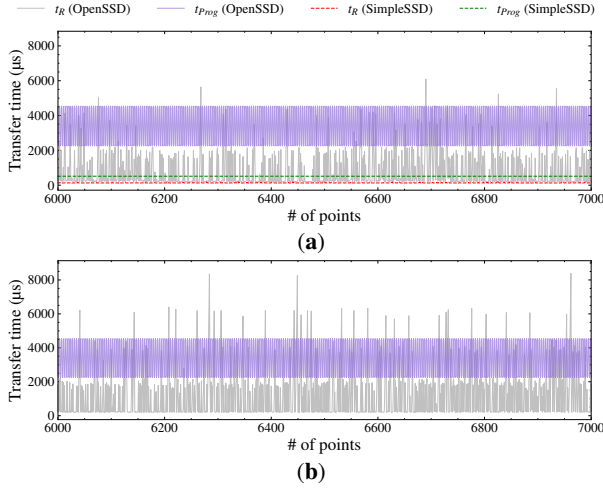


Fig. 4: NAND read/program I/O times of two different types of NAND (a) and (b) with $iodepth=8$. The data is zoomed in to show the 6000-7000 range for clarity.

B. Real-Life NAND I/O Characteristics in SSDs

To show the degree of deviation of real-life hardware from a theoretical simulation, we measured NAND read/program I/O request times of two different NAND modules from (a) SK Hynix and (b) Toshiba using the OpenSSD platform [23]. The specification of said NAND modules can be seen in Table I. The measurements were taken using the `randread` and `randwrite` benchmarks of `fiio` [30] with an I/O unit of the NAND page size. The specifications of the OpenSSD platform and host workstation are in Tables III and IV.

TABLE I: Specifications of NAND flash modules used.

Manufacturer	Capacity	Parallelism Setup	NAND Flash Page Size
(a) SK Hynix	1 TiB	4 Channel,	16 KiB
(b) Toshiba	256 GiB	8 Way	

Measurements of the time between when the NAND I/O request is issued to the low-level NAND flash controller by the SSD firmware, and when the firmware receives confirmation of request completion by the NAND flash controller were taken. For comparison with the SSD simulation platform, results from SimpleSSD was measured using its built-in I/O request generating benchmark, with simulation parameters taken from NAND (a) specifications. As SimpleSSD experiments used only NAND (a) parameters, the results are shown exclusively in figures based on that setup.

Fig. 3 shows the results with one outstanding I/O request, and reveals the differing performance characteristics of real and simulated NAND. Comparing NAND (a)'s read and program latency from OpenSSD and SimpleSSD shows that the real life NAND latency measurements are higher than the simulated results. Such results were observed as the main contributor to NAND latency is the provided NAND read/program latency parameters in the simulator.

With a workload with higher outstanding I/O requests, deviation from the simulation results exacerbates, as seen in Fig. 4 which shows the results with eight outstanding I/O requests.

TABLE II: Standard deviation of NAND read (t_R) and program (t_{Prog}) times of NAND (a), (b), and SimpleSSD's simulation, with $iodepth=1$ and $iodepth=8$. Units are in microseconds.

NAND Type	$iodepth=1$		$iodepth=8$	
	t_R	t_{Prog}	t_R	t_{Prog}
(a)	1.1	37.61	974.16	1110.91
(b)	0.89	3.19	1374.84	1107.97
SimpleSSD	0	0	11.1	0

Table II shows the disparity of real and simulated NAND latency variance, with simulated NAND latency staying close to the given latency parameters, while real NAND latency's standard deviation grows to the thousand of microseconds. While SimpleSSD does show small amounts of deviation for t_R due to its timeline scheduling of NAND I/O, it is not able to cover the extent of actual deviation.

Such a difference in latency between real and simulated NAND I/O is due to the low-level NAND flash controller and SSD firmware overhead not being taken into consideration in the simulations. These additional factors that add additional latency therefore act as blind spots in the simulation accuracy.

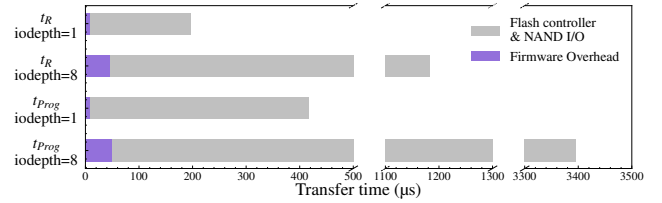


Fig. 5: Breakdown of NAND (b)'s average t_R and t_{Prog} .

Fig. 5 demonstrates the significance of these blind spots by breaking down the average NAND I/O time of NAND (b). The breakdown shows the presence of flash controller and firmware overhead, as well as a positive correlation with outstanding I/O requests and NAND I/O latency. This trend is further verified by the latency values documented by OpenSSD paper [23] showing the same trends as seen in our experimental results.

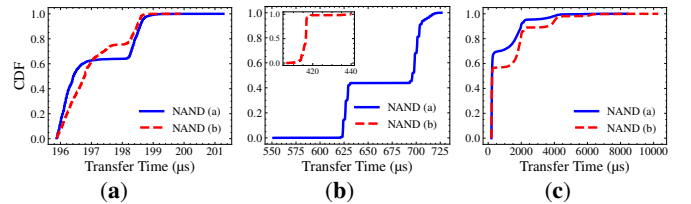


Fig. 6: NAND I/O latency Cumulative Distribution Function (CDF) of two different types of NAND in different workloads (a) `randread`, $iodepth=1$, (b) `randwrite`, $iodepth=1$, (c) `randread`, $iodepth=8$.

A deeper analysis of the real NAND results reveals the two NAND types also show different latency characteristics between each other. To demonstrate, Fig. 6 shows NAND I/O latency CDF graphs of the three previous `fiio` benchmarks. Even with benchmarks where both NAND types show

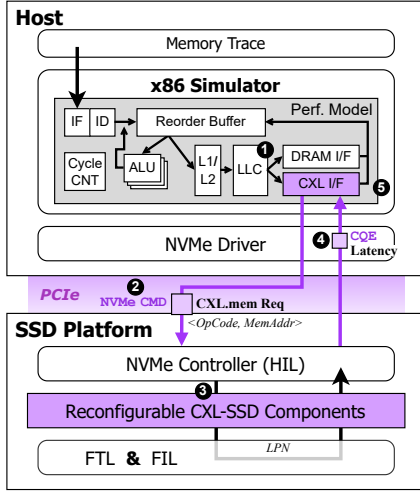


Fig. 7: Architecture of OPENCXD and its execution flow.

overlapping ranges of latency values, each NAND module shows differing distributions of latency for all benchmarks. Plus, returning to Fig. 3, the NAND (b) interestingly shows a brief spike in latency of up to $440\mu s$ before returning to near-median values, an effect that cannot be captured by simulators. **Bridging Simulation and Reality:** Our findings suggest that while NAND flash latency specifications are useful, *they do not fully capture the nuances of per-request performance in real-world scenarios*. In other words, although SSD simulators are effective for estimating average performance and extracting architectural insights, they often diverge from real behavior when used to evaluate fine-grained, real-time performance at the request level. This gap becomes especially critical in the context of CXL-SSDs, where the device functions more like memory than traditional block storage.

To address this issue, we explore the integration of the OpenSSD hardware platform with a host-side x86 simulator to construct a hybrid evaluation environment for CXL-SSDs. By *combining the cycle-level accuracy of x86 simulation with the realism of actual SSD hardware execution*, this approach enables the development and evaluation of CXL-SSD internal software components with both practical feasibility and performance characteristics that closely reflect real behavior. Building on this foundation, we propose **OPENCXD**, the first real-device-guided CXL-SSD simulation platform.

IV. OPENCXD: PROPOSED SYSTEM

A. Architectural Overview and Operation Flow

The overall architecture consists of two main components: *x86 simulator* and *SSD platform*. The *x86 simulator* provides cycle-accurate simulation of the entire memory hierarchy and replays memory traces derived from the target workload to reconstruct its instruction execution flow. It models detailed behaviors of the multi-layered cache hierarchy and the memory interface, allowing simulation of scenarios where Last-Level Cache (LLC) misses trigger memory accesses to either the main system memory or CXL memory. OPENCXD leverages this capability by intercepting LLC misses and, when the target address falls within a CXL-mapped range, redirecting

the memory request to the SSD platform. The *SSD platform* is built on an OpenSSD platform [23], which replicates the hardware architecture of a commercial SSD, incorporating a SoC controller, DRAM, and NAND flash chips. The SoC runs a SSD firmware, including the HIL, FTL, and FIL (§II-A).

As illustrated in Fig. 7, the execution flow of OPENCXD proceeds as follows: ① When an LLC miss occurs, the x86 simulator checks whether the missed address falls within the memory-mapped region assigned to the CXL-SSD. ② If it does, the request is encapsulated into a custom NVMe command that explicitly encodes the CXL.mem access semantics for OPENCXD, and issued to the SSD platform via NVMe passthrough [31]. ③ On the SSD side, the NVMe controller fetches the command and performs the corresponding CXL-SSD operation (e.g., write logging) based on the embedded request information. During this time, the x86 simulator pauses its cycle count. ④ The total device latency is measured on the SSD and returned to the host by embedding it in a reserved field of the Completion Queue Entry (CQE) [12]. ⑤ The NVMe driver extracts this latency from the CQE and reports it to the x86 simulator, which adds the CXL.mem interface overhead and integrates the total latency into its cycle count with resuming its cycle count progression.

B. x86 Simulator with CXL-SSD Integration

dword	description			
0	CID	P	Reserved	F opcode
1	Namespace ID			
2	Reserved			
3	Reserved			
4	Metadata Pointer			
5	Reserved			
6	PRPEntry#1 (not used)			
7	PRPEntry#2 (not used)			
8	Reserved			
9	Reserved			
10	Reserved			
11	CXL Memory Address			
12	Load or Store			
13	Reserved			
14	Reserved			
15	Reserved			

(a) NVMe Command (SQE)

□ NVMe Standard
■ OpenCXD Extension

※ Device Op Latency = NAND I/O Latency + Op Overhead

dword	description			
0	R	SCT	Op Overhead	R
1	Device Op Latency			
2	SO ID	CO ID		
3	Status Field	P	CID	

(b) NVMe CQE

Fig. 8: NVMe command and CQE for OPENCXD.

CXL Memory Request over NVMe: We extend the memory subsystem of the x86 simulator with a custom CXL.mem path that integrates with the SSD platform. This integration allows memory requests targeting the CXL-SSD region to be redirected as NVMe commands to the device. Specifically, the x86 simulator distinguishes memory requests by inspecting the target address of each 64-byte cacheline access. If the address falls within a CXL-SSD-mapped region, the simulator constructs a custom NVMe command that embeds the memory address and opcode of the memory request (see Figure 8(a)). Note that NVMe's data payload transfer is intentionally disabled [25], as the x86 simulators, including MacSim [13], typically do not model and process actual data payloads.

Device-in-the-Loop Timing Integration: Once the command is constructed, the x86 simulator issues it to the SSD platform via NVMe passthrough [31], then pauses its cycle progression and waits. Upon receiving the command, the SSD controller starts measuring the latency of the CXL-SSD operation. When the operation completes, the controller stops the timer, embeds the measured latency into a reserved field of the CQE, and returns the CQE to the host. To support fine-grained performance analysis and context switches used in CXL-SSD

optimizations, the controller also extracts and reports CXL operation overhead separately from the total device latency (see Figure 8(b)). Then, the simulator resumes execution by adding the CXL.mem interface overhead to the device-measured latency and converting the total delay into cycles based on its internal clock frequency.

To model the CXL.mem interface delay, OPENCXD applies a configurable overhead value. As prior work has shown [10], the CXL.mem interface delay is generally stable; for instance, SkyByte [11] assumes a fixed latency of 40 ns, which we also adopt in our evaluation. This delay parameter in OPENCXD can be adjusted to emulate different CXL interface designs.

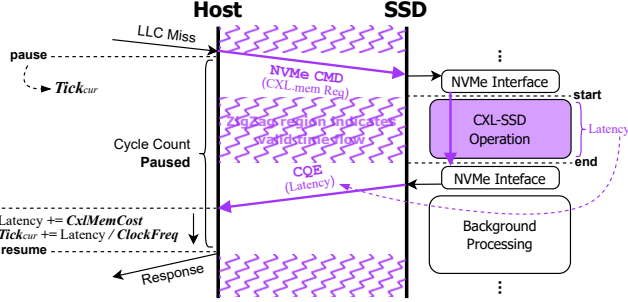


Fig. 9: Timing flow of a CXL memory request in OPENCXD.

As shown in Fig. 9, OPENCXD enables accurate *device-in-the-loop* integration by suspending the x86 simulator during each CXL-SSD access and incorporating measured firmware-level latencies. This approach faithfully models CXL-SSD timing behavior while excluding NVMe communication overhead, which would not be present in real CXL-SSDs.

C. SSD Platform with CXL-SSD Firmware Support

To reproduce the behavior of CXL-SSDs, we implement key software components described in Section II-B on the SSD platform. These include a *Write Log*, a *Data Cache*, and a *Log Index* with log compaction support. In a conventional SSD, incoming NVMe commands are typically handled by the HIL using a block I/O handler. In contrast, OPENCXD defines a new handler tailored for CXL-SSD operations. This custom handler first extracts the memory address and opcode (e.g., Read, Write) from the given CXL.mem memory request. It then performs the corresponding operation based on the opcode. For example, in the case of a write, the handler appends the data to the current index of the *Write Log*. If the *Data Cache* already contains the NAND page that includes the target cacheline, the handler updates the relevant cacheline offset within that cached page. Then, the *Log Index* is updated to record the cacheline’s location, as illustrated in Fig. 2.

D. Concurrent Access Modeling and Future Extensions

OPENCXD in its current form can simulate concurrent CXL-SSD access on the x86 simulator level. By replaying workload traces collected from 8 Skylake cores with 3 threads each—naturally containing interleaved CXL.mem accesses—OPENCXD reflects much of the timing behavior of concurrent requests from a host system. While the current design effectively models host-level concurrency, it focuses on

sequential request processing within the SSD platform when interfacing through NVMe passthrough via ioctl. This design choice simplifies the integration between the x86 simulator and the SSD platform, ensuring consistent timing control and reproducibility during trace replay. However, it also means that simultaneous in-device processing paths, such as overlapping NAND I/O operations or internal command pipelining, are not exercised. This limitation may lead to an underestimation of performance in workloads with a high consecutive cache miss ratio in the CXL-SSD. Enhancing this aspect is planned for future work, which will enable OPENCXD to more extensively reflect performance characteristics of highly parallel CXL-SSD architectures.

V. EVALUATION

A. Evaluation Setup

Implementation: We implement OPENCXD¹ by combining MacSim [13] with OpenSSD [23]. Parts from SkyByte’s implementation of MacSim were modified and used in OPENCXD as well. For the host-side simulation, we use the latest master branch of MacSim extended with CXL.mem support, modeling 8 Skylake CPU cores with latency parameters. To evaluate context switching behavior, we configure up to 3 threads per core. On the device side, we adopt the latest OpenSSD platform, DaisyPlus [32], and extend its firmware with state-of-the-art CXL-SSD components described in §II-B. The specifications of both the DaisyPlus platform and the host system are listed in Tables III and IV.

TABLE III: Specifications of the OpenSSD platform.

SoC	Xilinx Zynq UltraScale+ ZU17EG, with ARM Cortex-A53 Core
NAND Module	256GB, 4 Channel & 8 Way
Interconnect	PCIe Gen3 × 16 End-Points
DRAM	2GB LPDDR4 @ 2400MHz

TABLE IV: Specifications of the host system.

CPU	Intel(R) Core(TM) i7-14700K CPU @ 5.60GHz (28 cores)
Memory	32GB DDR5
OS	Ubuntu 24.04.2, Linux Kernel 6.11.0

Evaluation Methodology: We evaluate OPENCXD using memory traces from SkyByte [11], a state-of-the-art CXL-SSD study, which includes seven workloads: *bc*, *bfs-dense*, *d1rm*, *radix*, *srad*, *tpcc*, and *ycsb*. Each workload is executed for one million memory accesses, except for *bfs-dense*, which completes its full trace before reaching that threshold. Note that while SkyByte is also built on MacSim, its SSD backend is implemented using the SimpleSSD [15] simulator (§III). For a fair comparison, we modify SkyByte’s latency parameters to match the NAND and DRAM characteristics of our OpenSSD setup (see Table III). Both systems perform SSD data prefilling and host-side memory warm-up before executing the benchmarks.

B. Re-Evaluation of State-of-the-Art CXL-SSD Optimizations

Fig. 10 shows the average latencies of the performed benchmarks. A key difference between OPENCXD and SkyByte

¹<https://github.com/hschung1652/opencxd>

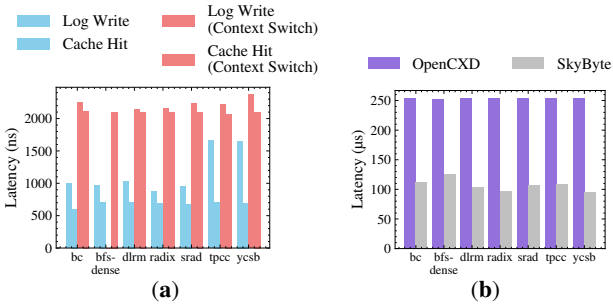


Fig. 10: Average latency of key CXL-SSD optimizations, with write log inserts and DRAM cache hits as seen in OPENCxD (a) and cache misses as seen in both platforms (b).

can be seen in the write log insert and SSD DRAM cache hit values, as seen in Fig. 10(a). SkyByte uses static parameters applied at compile-time to calculate these features, leading to write log insert and cache hit times to always be 640ns and 712ns respectively. However, OPENCxD experiences differing latencies for each workload performed, and some latencies of both operations go beyond the $2\mu\text{s}$ context switch threshold, which SkyByte uses to trigger context switches [11].

This shows that memory workloads with CXL-SSDs can show latency peaks well above average DRAM access times. As all I/O with CXL-SSDs involve DRAM load/store operations, they will be highly sensitive to DRAM performance penalties like latency spikes, requiring optimization to alleviate said penalties. Another difference seen is the SSD DRAM miss latency values in Fig. 10(b). By accounting for NAND controller and firmware overheads, OPENCxD shows $2.4\times$ higher average latency than SkyByte across all benchmarks.

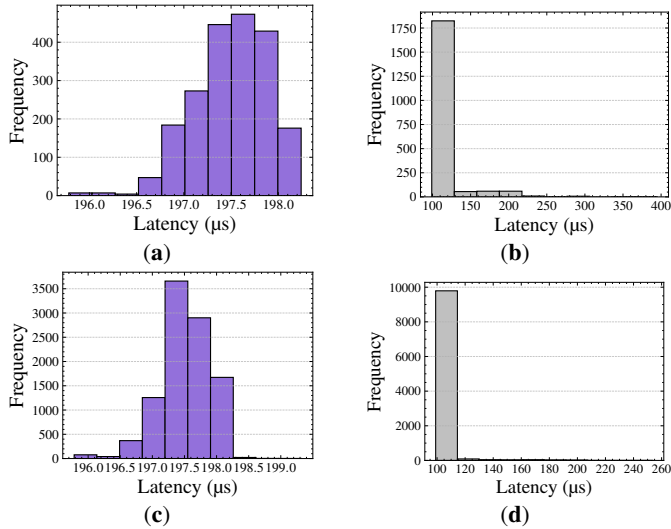


Fig. 11: Histograms of NAND read I/O latency during hit misses for *srad* ((a): OPENCxD, (b): SkyByte) and *ycsb* ((c): OPENCxD, (d): SkyByte).

A deeper analysis of the spread of latencies can be seen in Fig. 11. The histograms show SkyByte’s disproportionately high percentage of a single NAND read latency value being used, with *srad* and *ycsb* using the same value of $99.72\mu\text{s}$ 87.2% and 94.3% of the time respectively. SkyByte’s SSD sim-

ulation does take into account NAND I/O timeline scheduling. However, for the majority of cases, it is unable to replicate a realistic spread of latencies as seen in OPENCxD. Excluding finding an applicable timeslot for the NAND I/O to take place, SkyByte only performs mathematical calculations to apply the NAND latency [33]. Tracking these dynamic changes in latency is critical in evaluating CXL-SSDs, where performance is sensitive on a per memory request basis. Therefore, using a platform that can reflect these features like OPENCxD can offer valuable insights in the design of CXL-SSD optimizations.

Fig 12 reveals the consequences of the previously made observations in overall performance impact. In all workloads, OPENCxD required more CPU cycles to perform the amount of instructions required to perform one million memory accesses. This is in relation to the usage of context switching in an attempt to avoid the latency penalty of a NAND read. As said latencies are much higher in OPENCxD, context switching over 3 threads was not enough to completely hide the read latency. These results show that more threads are required to hide the high read latency via context switching, or new optimizations are required to cover NAND I/O.

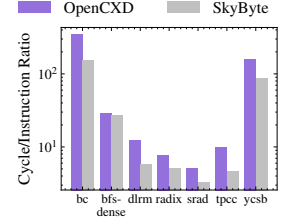


Fig. 12: Log-scale comparison of CPU cycles per completed instruction for both systems.

C. Time breakdown of CXL-SSD optimizations

TABLE V: Average and standard deviation of operation time related to CXL-SSD optimization. Units are in nanoseconds.

Workloads		Check DRAM Cache	Insert DRAM Cache entry	Check Write Log
<i>srad</i>	Average	37.02	32.04	170.86
	Stddev	29.44	29.93	54.57
<i>ycsb</i>	Average	36.31	34.93	183.2
	Stddev	29.79	29.59	30.03

Table V shows a breakdown of operation overhead of state of the art CXL-SSD optimizations run on the SSD controller. While overhead for DRAM cache operation on average are seen to be around 30ns, the standard deviation is also 30ns, showing a variance in operation overhead. The overhead for checking the write log index has notably higher operation overhead, showing a clear divide of operation characteristics within the CXL-SSD controller.

D. NAND Parallelism-Orientated Log Compaction

One of the key advantages of OPENCxD is its ability to reconfigure the firmware logic on the OpenSSD platform. This flexibility allows us to experiment with CXL-SSD-specific optimizations, particularly those that exploit NAND parallelism. As a case study, we redesigned the log compaction mechanism to leverage parallel NAND channels. Originally, log compaction processes NAND pages sequentially: it checks the first-level index to locate modified pages, loads them into memory if not cached, merges buffered

cachelines from the second-level index, and flushes the merged result to NAND (§II-B). In our parallel version, OPENCXD first scans and tracks all required NAND pages, batches the corresponding I/O requests, and then issues them simultaneously, enabling channel-level parallelism during compaction. The evaluation of this optimization can be seen in Fig. 13, where improvements of up to 8× is seen across all write log sizes. These results not only show the performance uplift of considering NAND parallelism, but also OPENCXD’s ability to demonstrate implemented optimizations in real hardware.

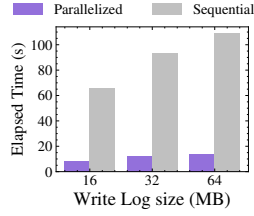


Fig. 13: Comparison of write log compaction based on exploiting NAND parallelism across different write log sizes.

VI. CONCLUSION

We present OPENCXD, a hybrid evaluation framework that combines cycle-accurate x86 system simulation with real SSD firmware execution on an OpenSSD platform. By bridging simulation and hardware, OPENCXD captures critical device-level behaviors, such as DRAM latency spikes and low-level NAND controller and SSD firmware overheads, that prior simulation-only approaches overlook. Our results highlight the necessity of real-device-guided evaluation for accurate analysis and design of CXL-SSDs.

ACKNOWLEDGMENTS

This work was partially supported by the National Research Foundation of Korea (NRF) grants funded by the Korea government (MSIT) (RS-2025-00564249 and RS-2024-00453929).

REFERENCES

- [1] Y. Kwon and M. Rhu, “Beyond the memory wall: a case for memory-centric hpc system for deep learning,” in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.
- [2] K. Derbyshire, “Memory Wall Problem Grows With LLMs,” *Semiconductor Engineering*, 2025. Last accessed: 2025-05-03.
- [3] Q. Zheng, J. Lee, D. A. Manno, and G. Grider, “Toward Standardized, Open Object-Based Computational Storage For Large-Scale Scientific Data Analytics,” in *Proceedings of the 8th International Parallel Data Systems Workshop (PDSW)*, 2023.
- [4] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, “AI and Memory Wall,” *IEEE Micro*, vol. 44, no. 03, 2024.
- [5] G. F. Oliveira, S. Ghose, J. Gómez-Luna, A. Boroumand, A. Savary, S. Rao, S. Qazi, G. Grignou, R. Thakur, E. Shiu, and O. Mutlu, “Extending Memory Capacity in Modern Consumer Systems With Emerging Non-Volatile Memory: Experimental Analysis and Characterization Using the Intel Optane SSD,” *IEEE Access*, vol. 11, 2023.
- [6] A. Badam and V. S. Pai, “SSDAlloc: hybrid SSD/RAM memory management made easy,” in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2011.
- [7] M. Jung, “Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD),” in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2022.
- [8] CXL Consortium, “Compute Express Link (CXL) Specification Revision 3.1,” <https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.1-Specification.pdf>, 2023. Accessed: 2025-05-29.
- [9] PCI-SIG, “PCI-SIG Specifications,” <https://pcisig.com/specifications>. Accessed: 2025-05-29.

- [10] D. Das Sharma, R. Blankenship, and D. Berger, “An Introduction to the Compute Express Link (CXL) Interconnect,” *ACM Computing Surveys*, vol. 56, no. 11, 2024.
- [11] H. Zhang, Y. Xue, Y. E. Zhou, S. Li, and J. Huang, “SkyByte: Architecting an Efficient Memory-Semantic CXL-based SSD with OS and Hardware Co-design,” in *Proceedings of the 2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025.
- [12] NVM Express Inc., “NVM Express Specification,” <https://nvmexpress.org/developers/nvme-specification>, 2011. Last Accessed: 2025-05-10.
- [13] Georgia Institute of Technology, “MacSim: A Heterogeneous Architecture Timing Model Simulator,” <https://github.com/gthparch/macsim>, 2025. Accessed: 2025-05-29.
- [14] The gem5 Community, “The gem5 Simulator System,” <https://www.gem5.org/>, 2025. Accessed: 2025-05-29.
- [15] CAMELlab, “SimpleSSD 2.0.12 Documentation,” <https://docs.simplessd.org/en/v2.0.12/>, 2020. Accessed: 2025-05-29.
- [16] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, “FlashSim: A Simulator for NAND Flash-Based Solid-State Drives,” in *Proceedings of the 2009 First International Conference on Advances in System Simulation (SIMUL)*, 2009.
- [17] S. Li, Y. E. Zhou, H. Ren, and J. Huang, “ByteFS: System Support for (CXL-based) Memory-Semantic Solid-State Drives,” in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.
- [18] Y. Zhan, H. Hu, X. Yang, S. Wang, Q. Cao, H. Jiang, and J. Yao, “RomeFS: A CXL-SSD Aware File System Exploiting Synergy of Memory-Block Dual Paths,” in *Proceedings of the 2024 ACM Symposium on Cloud Computing (SoCC)*, 2024.
- [19] Y. Wang, Z. Wang, F. Meng, Y. Wang, Y. Ou, L. Wu, W. Hong, X. Ge, and J. Cao, “A Full-System Simulation Framework for CXL-Based SSD Memory System,” *arXiv preprint arXiv:2501.02524*, 2025.
- [20] D. D. Sharma, “Transforming the Data-Centric World,” in *Flash Memory Summit (FMS)*, Aug. 2022. Keynote presentation.
- [21] Microchip Technology Inc., “XpressConnect™ PCIe® Gen 5 and CXL™ Retimer Family,” https://iotdesignpro.com/sites/default/files/component_datasheet/XpressConnect-PCIe-Retimers-Datasheet.pdf, Nov. 2020. Accessed: 2025-05-29.
- [22] D. Gouk, M. Kwon, J. Zhang, S. Koh, W. Choi, N. S. Kim, M. Kandemir, and M. Jung, “Amber: enabling precise full-system simulation with detailed modeling of all ssd resources,” in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.
- [23] J. Kwak, S. Lee, K. Park, J. Jeong, and Y. H. Song, “Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems,” *ACM Transactions on Storage*, vol. 16, no. 3, 2020.
- [24] SK hynix Inc., “SK hynix Completes Customer Validation of CXL 2.0-based DDR5,” <https://news.skhynix.com/sk-hynix-completes-customer-validation-of-cxl-based-ddr5/>, 2025. Accessed: 2025-05-29.
- [25] J. Park, C.-G. Lee, S. Hwang, S. Yang, J. Noh, W. Chung, J. Lee, and Y. Kim, “BandSlim: A Novel Bandwidth and Space-Efficient KV-SSD with an Escape-from-Block Approach,” in *Proceedings of the 53rd International Conference on Parallel Processing (ICPP)*, 2024.
- [26] J. Park, J. Lee, and Y. Kim, “ByteExpress: A High-Performance and Traffic-Efficient Inline Transfer of Small Payloads over NVMe,” in *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2025.
- [27] M. Kwon, S. Lee, and M. Jung, “Cache in Hand: Expander-Driven CXL Prefetcher for Next Generation CXL-SSD,” in *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2023.
- [28] SimpleSSD Project, “Platform Abstraction Layer (PAL) – SimpleSSD v2.0.12 Documentation,” <https://docs.simplessd.org/en/v2.0.12/v2.0/pal.html>, 2024. Accessed: 2025-06-02.
- [29] SimpleSSD Project, “PAL2.cc Source Code,” <https://github.com/SimpleSSD/SimpleSSD/blob/2.0/pal/old/PAL2.cc>, 2024. Accessed: 2025-06-02.
- [30] J. Axboe, “Flexible I/O Tester (fio),” <https://github.com/axboe/fio>, 2022. Accessed: 2025-05-29.
- [31] linux nvme, “libnvme: C Library for NVM Express on Linux,” <https://github.com/linux-nvme/libnvme/blob/master/src/nvme/ioctl.h>, 2025. Accessed: 2025-05-29.
- [32] CRZ.TECHNOLOGY, “Daisy+ OpenSSD,” <https://www.mangoboard.com/main/view.asp?idx=1056&cate1=9>. Accessed: 2025-05-29.
- [33] Zhang, Haoyang and Xue, Yuqi and Zhou, Yirui Eric and Li, Shaobo and Huang, Jian, “ftl.cc Source Code,” <https://github.com/platformxlabs/skybyte/blob/main/src/SkyByte-Sim/ftl.cc>, 2025. Accessed: 2025-06-02.