SkyByte: Architecting an Efficient Memory-Semantic CXL-based SSD with OS and Hardware Co-design

Haoyang Zhang*, Yuqi Xue*, Yirui Eric Zhou, Shaobo Li, Jian Huang

University of Illinois Urbana Champaign

{zhang402, yuqixue2, yiruiz2, shaobol2, jianh}@illinois.edu

Abstract—The CXL-based solid-state drive (CXL-SSD) provides a promising approach towards scaling the main memory capacity at low cost. However, the CXL-based SSD faces performance challenges due to the long flash access latency and unpredictable events such as garbage collection in the SSD device, stalling the host processor and wasting compute cycles. Although the CXL interface enables the byte-granular data access to the SSD, accessing flash chips is still at page granularity due to physical limitations. The mismatch of access granularity causes significant unnecessary I/O traffic to flash chips, worsening the suboptimal end-to-end data access performance.

In this paper, we present SkyByte, an efficient CXL-based SSD that employs a holistic approach to address the aforementioned challenges by co-designing the host operating system (OS) and SSD controller. To alleviate the long memory stall when accessing the CXL-SSD, SkyByte revisits the OS context switch mechanism and enables opportunistic context switches upon the detection of long access delays. To accommodate byte-granular data accesses, SkyByte architects the internal DRAM of the SSD controller into a cacheline-level write log and a page-level data cache, and enables data coalescing upon log cleaning to reduce the I/O traffic to flash chips. SkyByte also employs optimization techniques that include adaptive page migration for exploring the performance benefits of fast host memory by promoting hot pages in CXL-SSD to the host. We implement SkyByte with a CXL-SSD simulator and evaluate its efficiency with various data-intensive applications. Our experiments show that SkyByte outperforms current CXL-based SSD by 6.11×, and reduces the I/O traffic to flash chips by 23.08× on average. SkyByte also reaches 75% of the performance of the ideal case that assumes unlimited DRAM capacity in the host, which offers an attractive cost-effective solution.

I. INTRODUCTION

CXL-based solid-state drives (CXL-SSDs) have been presented as a practical and cost-effective approach towards expanding the memory capacity [54], [62], as current manufacturing technology has allowed SSDs to scale up to terabytes, and the cost of SSDs is significantly lower than DRAM [7], [26], [49], [58]. The CXL-SSD allows programs to use the SSD as main memory via load/store instructions in a transparent fashion. It enables byte-granular data access to SSDs via CXL protocols. Because of these enabled memory properties in SSDs, we also define CXL-SSD as memory-semantic SSD.

The CXL technology facilitates the use of flash-based SSDs as memory [7], [11], [54]. However, simply treating SSDs as an extension of host memory via CXL causes dramatic performance degradation and excessive CPU stalls (see §II-C). This is for three major reasons. First, the flash access latency

is several orders of magnitude higher than the host DRAM latency. Although the SSD has an internal DRAM cache, the capacity is relatively small (a few GBs in modern SSDs) [42], [57], its miss penalty is still determined by the long flash access latency. Second, due to the inherent properties of flash memory (i.e., out-of-place updates and garbage collection), the complexity of managing flash chips inside the SSD controller causes performance interference. For instance, the garbage collection (GC) of SSDs will postpone the read/write requests to flash chips until the GC is finished. The underlying SSD events can cause long host CPU stalls and unpredictable end-to-end performance. Third, although CXL technology enables byte-granular data access to SSDs, flash chips support only page-granular data access due to physical limitations [11], [20], [42]. The mismatch of data access granularity between CXL (byte-granular) and flash chips (page-granular) causes high I/O amplification and extra I/O traffic to flash chips.

To address the above challenges, we employ a holistic approach to develop an effective CXL-based SSD, named SkyByte, by co-designing the host OS and CXL-based SSD controller. We elaborate the key ideas of SkyByte as follows. Coordinated context switch for CXL-SSD. To hide memory access latency, processors typically employ out-of-order execution and issue multiple memory requests in parallel in the hope that there are sufficient non-memory instructions to fill the pipeline, while waiting for the response from the memory. This technique has been proven effective with the host DRAM, however, it fails to hide the long flash access latency of CXL-SSDs, unless the processor can examine an impractically large instruction window (for identifying sufficient instructions to hide the memory latency). In modern OS, if one thread needs to wait for a long SSD access, the OS can perform a context switch and select another thread to utilize the CPU core. Unfortunately, this context switch opportunity is missing for CXL-SSDs, because the OS cannot intercept the load/store memory instructions issued directly from the host CPU to the SSD device via the CXL protocol.

We revisit the OS context switch mechanism and develop a coordinated approach between the host OS and the SSD controller. To precisely track which instruction is blocked by long SSD delay, we extend the CXL.mem response packet format to encode a long-delay hint. When the SSD controller detects that a CXL memory request will suffer from a long delay, it responds to the host with this hint. The host CXL controller forwards this hint to the CPU core in the form of a

---

*Co-primary authors.

hardware exception triggered by the corresponding load/store instruction. Then, the exception handler calls the host OS scheduler to perform a context switch. SkyByte supports different policies for deciding when to trigger a context switch and which thread will be executed next (see §III-A).

CXL-aware SSD DRAM management. Since modern SSDs are primarily designed as block devices, they organize the DRAM cache in page granularity. However, for CXL-SSDs, our study in §II finds that most workloads access less than 40% of cachelines in more than 75% of pages. Caching the entire page in the SSD DRAM significantly wastes precious SSD DRAM space. This also leads to significant write amplification, as we need to write back the entire page to flash chips even though only a few cachelines of a page are dirty.

To bridge the gap between the cacheline-granular CXL interface and the page-granular flash chips, we structure the SSD DRAM into a cacheline-granular write log and a page-granular read-write cache. Write requests are served by the write log at cacheline granularity without first fetching the original page from flash chips. The read-write cache is managed in page granularity to exploit spatial locality, as we need to read an entire page from flash chips anyway. When the write log is full, SkyByte performs log compaction in the background to coalesce writes to the same page. This greatly reduces the write traffic to the flash chips and mitigates long flash write latency. For read requests, SkyByte looks up the write log and the read-write cache in parallel to locate the latest data with an efficient hash-based indexing mechanism (see §III-B).

Since the SSD DRAM capacity is limited, we leverage the host memory to expand the SSD DRAM capacity by enabling adaptive page migrations in the background. SkyByte identifies hot pages in the SSD DRAM and performs page migrations transparently (see §III-C). SkyByte ensures data consistency during page migrations by employing a promotion buffer in the host bridge developed in prior studies [7]. Upon the completion of a page migration, the corresponding page table entry will be updated to reflect the new memory address.

We implement SkyByte with a CXL-SSD simulator based on MacSim [41] and SimpleSSD [21]. We extend MacSim to simulate context switches on each CPU core and modify its memory interface to simulate the CXL.mem. The CXL memory requests are sent to the SSD that has the write log, the data cache, and the flash translation layer (FTL). We evaluate SkyByte with data-intensive workloads (see Table I). For each workload, we capture the instruction traces of each thread using PIN [30] and replay the multi-threaded traces in our simulator. Our evaluation shows that SkyByte outperforms state-of-the-art CXL-SSDs by 6.11×, and reduces the I/O write amplification to the flash chips by 23.08× on average. SkyByte also achieves 75% of the performance of the ideal case assuming unlimited host DRAM capacity, demonstrating its benefit on cost-effectiveness. In summary, we make the following contributions:

- We examine the performance bottlenecks of CXL-SSDs, and identify that they are caused by excessive CPU stalls due to long CXL memory access latency, and the access granularity mismatch between the CXL interface and the flash memory.
- We propose SkyByte, which employs an OS and hardware co-design approach to hide the flash access latency of CXL-based SSDs with a coordinated context switch.
- We re-architect the SSD DRAM cache with a write log and a read-write cache to bridge the gap between the page-granular flash accesses and the byte-granular CXL memory accesses.
- We implement SkyByte in a CXL-SSD simulator to accurately simulate the interplay among the CXL-SSD, the multi-core processor microarchitecture, and the OS scheduling.
- We evaluate the effectiveness of SkyByte with various data-intensive workloads and sensitivity analysis, showing that SkyByte is a practical and cost-effective approach.

II. BACKGROUND AND MOTIVATION

A. CXL and Memory Expansion

The Compute Express Link (CXL) [18] is a new interconnect standard built on PCIe 5.0 physical interface. It can construct a unified and coherent memory space and enable high-speed communication across different types of processors, memory, and accelerators. CXL has been rapidly gaining industry adoption and is on track to become a primary interconnect. CXL defines three protocols that include CXL.io, CXL.cache, and CXL.mem for different purposes. CXL.io is functionally equivalent to the traditional PCIe protocol. CXL.cache enables cache coherence between interconnected devices. CXL.mem enables the device's local memory to be directly accessed by the host CPU via load/store instructions. With these protocols, CXL supports three primary device types: Type-1 is for devices to access the host memory in a cache-coherent manner (only CXL.cache enabled), such as specialized accelerators like NICs; Type-2 is for devices and the host CPU to access each other's memory with cache coherence (both CXL.cache and CXL.mem enabled); and Type-3 is for devices that allow the host CPU to access and cache its memory at cacheline granularity (only CXL.mem enabled), such as memory expander devices.

In this paper, we use SSD as a Type-3 CXL device. The entire SSD is exposed as host-managed device memory (HDM). The SSD memory space is mapped to the host's physical memory space. The CXL.mem protocol allows the host CPU to directly access the SSD via cachable load/store instructions.

B. Architecture of CXL-based SSDs

A simple approach to building a CXL-SSD is to directly connect CXL with a byte-addressable SSD by having a few simple changes to the SSD controller, as described in prior studies [12], [32], [62]. We show its architecture in Figure 1.

Figure 1 (a) shows the memory mapping with a CXL-SSD. Upon booting, the host initializes the CXL-SSD by mapping its logical memory space into the system physical memory space. The CXL memory or HDM is exposed to the OS as normal physical memory. It is CPU-cacheable and accessible with load/store instructions. However, it possesses different performance attributes compared to the host DRAM. Therefore,
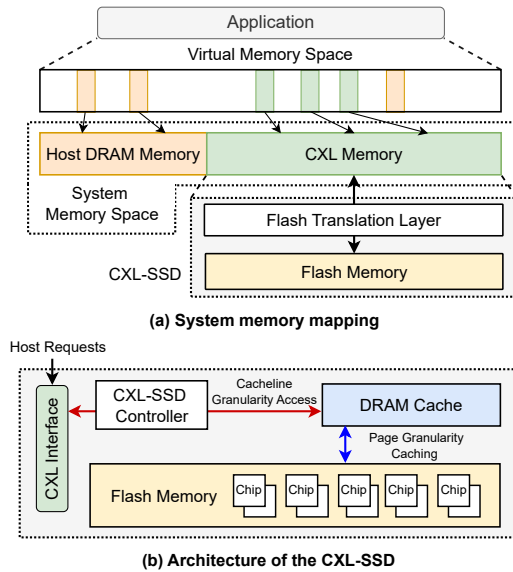
(a) System memory mapping

(b) Architecture of the CXL-SSD

Fig. 1: System architecture of CXL-based SSD (CXL-SSD).

Fig. 2: End-to-end execution time of running different workloads using DRAM vs. CXL-SSD.

Fig. 3: Latency distribution of DRAM vs. CXL-SSD.

the entire system memory can be considered as a heterogeneous memory system. The host OS remains responsible for managing the memory placement and the virtual-to-physical memory mapping. And the Flash Translation Layer (FTL) of the SSD handles the address translation from the logical page address (LPA) to the physical page address (PPA) of flash memory.

Figure 1 (b) shows the architecture of the CXL-SSD. When an application generates a memory access to the CXL-SSD, the CXL home agent will send a message via the CXL.mem protocol. The SSD controller will then parse the message to extract the memory request and serve the data by coordinating with the SSD firmware. To support cacheline (64B) access granularity, the controller utilizes the DRAM cache inside the SSD to serve the memory requests from the host. The SSD DRAM will cache the data from flash chips at page granularity.

C. Challenges of Using CXL-Based SSDs

Although the CXL technology offers a great opportunity for the wide adoption of memory-semantic SSDs, the current OS and processor architecture does not work well with CXL-SSDs out of the box. Naïvely treating CXL-SSDs as conventional DRAM memory will lead to severe performance degradation.

To understand this issue, we study various data-intensive applications (see Table I) and examine their performance when allocating all their data (1) in a CXL-SSD device as described in §II-B and (2) in the host DRAM. For each program, we launch four threads on four cores without hyperthreading. We use PIN [30] to collect the instruction and memory traces, and replay the traces in a cycle-accurate simulator (see §V for details) to quantify the microarchitectural performance impact of using CXL-SSDs. We summarize our key insights as follows.

Long tail latency. Figure 2 shows the total execution time of the workloads with host DRAM and CXL-SSD, respectively. These workloads perform 1.5–31.4× worse with CXL-SSD than with DRAM due to the long flash memory access latency even
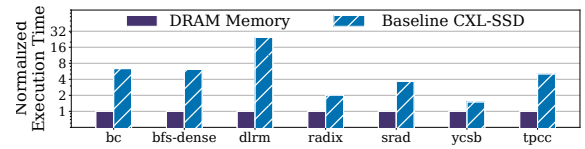
with SSD internal DRAM cache. Figure 3 shows the off-chip memory access latency distribution of DRAM and CXL-SSD. Due to space limitations, we only show four representative workloads (all workloads have similar patterns). While more than 90% of the CXL-SSD memory requests are within 200 ns thanks to the SSD DRAM cache, the tail latency can be as high as hundreds of μs when a flash read/write happens due to the SSD DRAM cache miss. The latency will be even higher (e.g., a few milliseconds) when garbage collection is triggered.

Excessive processor pipeline stalls. The tail latency of CXL-SSD causes severe processor pipeline stalls, leading to both performance degradation and underutilization of CPU and SSD bandwidth. We quantify the impact of pipeline stalls by analyzing compute vs. memory boundedness, following Intel's Vtune profiling tool [29]. We define that a clock cycle is *bounded by memory* if no instructions except memory operations are executing in this cycle (i.e., the pipeline is stalled by memory) and *bounded by compute* otherwise. Figure 4 quantifies the percentage of cycles bounded by memory or compute. The portion of memory-bounded cycles grows from 62.9%–98.7% with DRAM to 77%–99.8% with CXL-SSD. Although modern processors employ techniques such as OoO and multi-level caches to hide memory latency and exploit memory parallelism, they are less effective for hiding the long flash access latency, causing severe pipeline stalls.

Even worse, the long pipeline stalls lead to memory bandwidth underutilization of the CXL-SSD device. Although we can use more cores to improve the bandwidth, this will also lead to more severe compute underutilization as more cores are being stalled. This is because the processor cannot keep enough in-flight memory requests to saturate the available SSD bandwidth. For example, to saturate a single DDR5 channel with a bandwidth of 32GB/s and 70 ns latency for each 64B cache line,
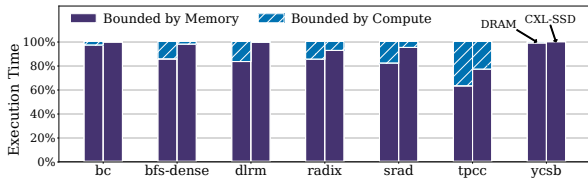
Fig. 4: Execution boundedness breakdown of various workloads with DRAM vs. CXL-SSD.
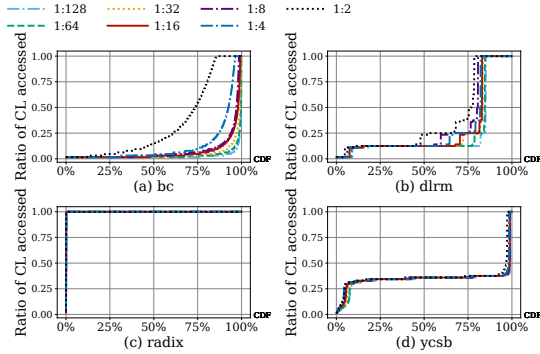

Fig. 5: Locality distribution of all pages read from flash chips into the SSD DRAM cache. The legend "1:n" means the workload's memory footprint is n× larger than the SSD DRAM cache. The y-axis is the percentage of cache lines accessed in each page.

Fig. 6: Locality distribution of all pages flushed to flash chips from the SSD DRAM cache. The legend "1:n" means the workload's memory footprint is n× larger than the SSD DRAM cache. The y-axis is the percentage of dirty cache lines in each page.

we need to issue at least $70 \times 32/64 = 35$ concurrent memory requests. To hide the flash access latency (3 μs read latency for state-of-the-art Z-NAND [52]) assuming a bandwidth of 16 GB/s for PCIe 5.0 x4, we need $3000 \times 16/64 = 750$ memory requests, which is impractical for today's processor.

Access granularity mismatch between CXL interface and flash memory. To hide the long flash access latency, modern SSDs typically employ an internal DRAM cache managed at flash page granularity, as they are designed for the block interface, and flash chips support only page-granular access [11], [20], [42]. Thus, the current SSD DRAM cache design becomes significantly less effective for the CXL-SSD due to the access granularity mismatch between the CXL interface (64B cache line) and the flash memory (4KB page or even larger).

We quantify the memory access patterns of different workloads in Figure 5 and Figure 6. Many workloads only access less than 40% of the cache lines in more than 75% of pages. This causes two problems. First, as we cache an entire page in the SSD DRAM, the DRAM capacity is significantly wasted as most cache lines in the page are not accessed. Second, even if we only write a few cache lines in a page, we still need to write the entire page to the flash memory, which leads to write amplifications and shortens the SSD lifetime. Enlarging the SSD DRAM capacity has limited benefits unless the DRAM is sufficiently large to hold the entire working set of workloads.

III. DESIGN OF SKYBYTE

SkyByte consists of three major components: (1) the coordinated context switch mechanism based on the detection of long SSD access delays (§III-A); (2) a cacheline-granular
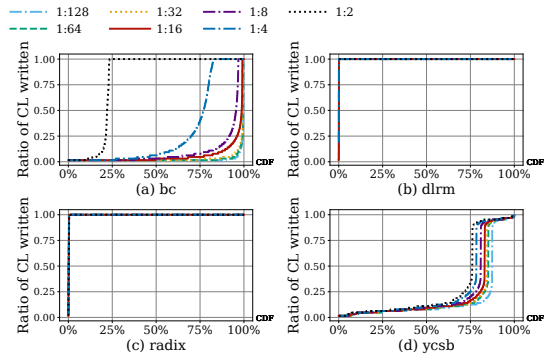
log-structured memory in SSD controller, for bridging the gap between the byte-granular CXL interface and the page-granular flash chips (§III-B); (3) an adaptive page migration mechanism that leverages the host memory to expand SSD DRAM by migrating hot pages to the host in a transparent and consistent manner (§III-C). We discuss each of them as follows.

A. Coordinated Context Switch Mechanism

When a thread encounters a long CXL-SSD access caused by SSD DRAM cache miss, we can context switch to another thread to better utilize the CPU core. However, the SSD device has no knowledge about the microarchitectural status of the host CPU, such as which core triggers a missed memory access and whether this load is speculative. Similarly, the host CPU does not know whether a memory access is a hit/miss in the SSD DRAM cache. Neither CPU nor SSD can by itself decide whether to trigger a context switch. Therefore, to enable context switch on long CXL-SSD memory stalls, we coordinate between host OS and SSD controller.

To enable a coordinated context switch, we need to decide (1) when to trigger the context switch; (2) how to conduct the context switch; and (3) what are the policies for the context switch. To address these questions, we first present the coordinated context switch procedure. Then, we discuss policies for deciding when to trigger a context switch and which thread is executed next. Finally, we discuss the hardware and software modifications needed for this mechanism.

Context switch procedure. We show the context switch procedure with an example of a CXL memory read from the host CPU in Figure 7. As writes are buffered in the write log (see §III-B), they do not need to trigger context switch.

(C1) *Sending CXL.mem request message with tracking information.* The host CPU sends a CXL.mem `MemRd` request message to the SSD controller. By default, the CPU maintains microarchitectural status, including the miss status handling registers (MSHRs) of the shared LLC, for tracking which load/store instruction in which core is waiting for the response of this memory request. The MSHRs also perform memory access coalescing, so a memory request may be associated
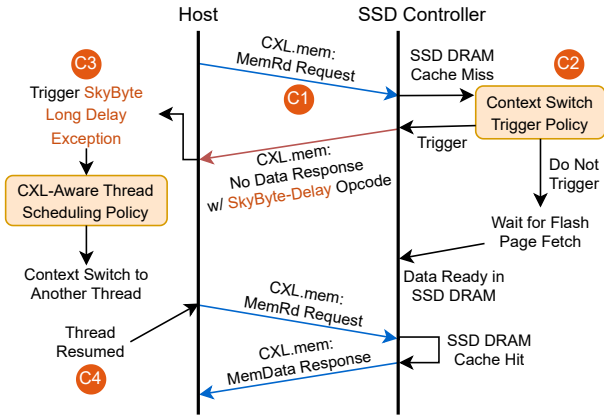
Fig. 7: The procedure of coordinated context switch in SkyByte.

Fig. 8: No Data Response (NDR) message format and opcode definitions in CXL.mem. SkyByte uses one of the reserved opcodes (shaded in green) to indicate a long access delay.

Opcode	Description	Encoding
Cmp	Completions for Writebacks, Reads, and Invalidates	000b
Cmp-S, Cmp-E, BI-ConflictAck	Cache coherence-related opcodes for CXL.cache	001b, 010b, 100b
SkyByte-Delay	Indication from the SSD to the Host for Long Access Delay	111b
Reserved	Other reserved opcodes	Others

with multiple instructions from different cores if they request for the same cache line. The CXL controller tracks all the memory requests between the host CPU and the SSD via the CXL.mem `MemRd` message (see Figure 8).

C2 *Sending context switch request with extended CXL.mem No Data Response Message.* Upon an SSD DRAM cache miss, the SSD controller starts to fetch the page from the flash. It will determine whether or not to send a context switch request to the host OS based on an estimated access latency (as discussed later in the conetxt switch trigger policy). The SSD controller sends the context switch request via a No Data Response (NDR) message (one type of the slave-to-master (S2M) message). The NDR message indicates the completion of a CXL memory request without returning any data to the host CPU. As shown in Figure 8, the `MemRd` message includes a 16-bit tag [18] for each CXL.mem transaction. SkyByte extends the NDR message specification by introducing a new opcode called `SkyByte-Delay`. This opcode indicates that the corresponding `MemRd` request will suffer from a long access delay (e.g., an SSD DRAM cache miss).

C3 *Triggering context switch with hardware exception.* SkyByte leverages the existing hardware exception mechanism in modern CPUs to precisely track which load/store instruction in which core should trigger a context switch. SkyByte defines a new *SkyByte Long Delay Exception*. Once the host CPU receives the `SkyByte-Delay` NDR message from the CXL controller, it looks up the LLC MSHR entry of this memory request and traverses the upper-level cache hierarchy (e.g., L1 and L2) to find all uncommitted memory instructions waiting for this response. When any of these instructions enter the retire stage, it will trigger the SkyByte Long Delay Exception (similar to how a load/store instruction triggers a Page Fault Exception) on the corresponding core. The address of this instruction will be saved upon the context switch, such that when the thread is switched back, it will resume from this instruction and re-issue this memory access to the CXL-SSD.

Such a design eliminates false-positive context switches, where a load/store triggers a context switch but is later squashed, at no extra hardware cost, since modern processors by default delay the exception handling to the retire stage. For example,

speculative load/store and hardware prefetch will not trigger any exception even if they miss in the SSD DRAM.

SkyByte installs a special handler for the SkyByte Long Delay Exception in the x86 interrupt descriptor table (IDT). The exception handler invokes a CXL-aware thread scheduling policy to decide which thread is executed next and performs the context switch, which will be discussed later.

C4 *Resuming the original thread.* When the original thread is scheduled back, it will resume from the previously missed memory instruction that triggered the SkyByte Long Delay Exception. This memory access is then issued again to the CXL-SSD. If the replayed instruction hits in the SSD DRAM, the data will be returned with a CXL.mem `MemData` response.

When a thread is context-switched, all its pending load/store instructions are squashed. However, the MSHRs in the cache hierarchy may or may not be freed depending on the implementation [25], [47]. This can cause severe MSHR contention between threads given the long flash access latency. In the worst case, a thread performs accesses to 64 cache lines in multiple 4KB pages, all of which miss in the SSD DRAM cache. These requests will occupy at least 64 MSHRs for a few microseconds, easily exhausting all the MSHRs. To address this issue, we free the MSHR entry as soon as the corresponding instruction squashes. Since this approach can also benefit host DRAM accesses, we enable it in SkyByte by default.

When a thread is scheduled again after it has been context-switched away a while ago, it may trigger the same SSD DRAM cache miss again if the requested page has already been evicted due to a cache conflict. This is less of a concern, since the LRU eviction policy in the SSD DRAM cache already prevents the requested page from being evicted for most of the time. For instance, in our experiments with various data-intensive applications, we did not observe a page being evicted before the original thread resumes execution and accesses it.

Context switch trigger policy. Upon an SSD DRAM cache miss, the SSD controller can choose to either trigger a context switch or let the host CPU wait for the data. Intuitively, if the context switch overhead is smaller than the CXL-SSD access delay, we can perform a context switch to hide the delay.

SkyByte uses a *threshold-based policy* to decide whether a context switch should be triggered. The SSD controller estimates the latency of fetching the requested page from flash chips. If the estimated latency is higher than the threshold, a context switch will be triggered. The threshold can be tuned

Algorithm 1 Threshold-based context switch trigger policy.

```
1: function SHD_CTX_SWTC(req, threshold, read_lat, write_lat, erase_lat)
2:     PPA = translate_address(req);
3:     queue = get_channel_queue(PPA);
4:     num_read, num_write, num_erase = queue.get_counters();
5:     est_lat = read_lat * (num_read + 1) + write_lat * num_write
6:         + erase_lat * num_erase;
7:     return est_lat > threshold;
8: end function
```

based on the host context switch overhead.

We show the details of the threshold-based context switch trigger policy in Algorithm 1. SkyByte first looks up the FTL mapping table to get the physical page address (PPA), which determines which flash channel this request will be issued to (Line 2–3). It employs a simple approach to estimate the flash access latency by querying the corresponding flash controller's queue status, i.e., the number of requests placed in the queue (Line 4). Typically, the requests in the channel queue will be served in FIFO order [44]. Therefore, similar to the approach studied in prior work [38], [60], SkyByte can accurately estimate the delay of the request by summing the latency of all requests in the queue (Line 5–6). If the estimated delay is longer than the threshold, SkyByte will trigger a context switch (Line 7). If a request is blocked by an ongoing garbage collection (GC), SkyByte will immediately trigger a context switch, as GCs typically last for milliseconds. While the GC process will block the issuing of requests in the queue, its impact is already considered in the latency prediction algorithm by querying the flash channel queue status.

To set an appropriate threshold, we can measure the average overhead of context switches of the host CPU. Figure 9 shows the performance of various thresholds for representative workloads. Since the flash page read latency (3 μs by default) of our SSD is longer than the regular context switch overhead (2 μs, as examined with the hardware setup described in Table II), we set the threshold to 2 μs. In practice, the threshold can be tuned empirically for different CPUs and system configurations. SkyByte allows the host OS to configure it.

OS support for CXL-aware context switch. To enable the Long Delay Exception, we install a new exception handler into the x86 IDT. When the exception is raised, the exception handler yields the CPU resources owned by the current thread. The system scheduler then decides the thread to run next based on the predefined policy. The yield thread is re-enqueued back to the run queue in OS, allowing it to be scheduled again later.

We explore three scheduling policies for the scheduler to pick the next runnable thread in Linux OS: *(1) Round-Robin (RR) policy*, in which the threads take turns to execute; *(2) Random policy*, in which a thread is chosen randomly to execute next; *(3) Complete Fairness Scheduler (CFS) policy* [2], in which it prioritizes the thread which has the shortest received execution time for ensuring a fair share of CPU cycles among threads.

We evaluate the performance of these policies and show the execution time breakdown in Figure 10, following the same definition of boundedness in Figure 4. The three policies deliver similar performance. This is because all the threads are mainly bounded by memory I/O, and all three policies allow the threads
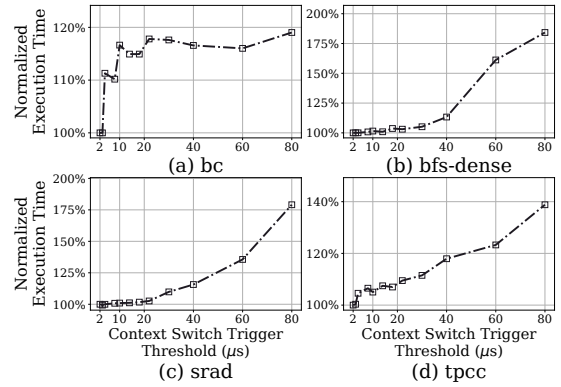

Fig. 9: Impact of the thresholds defined in the in the coordinated context switch trigger policy.
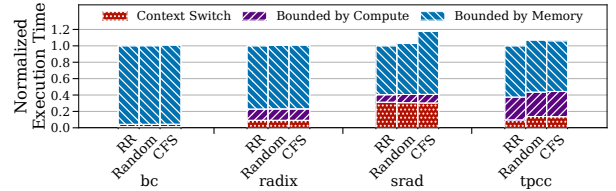

Fig. 10: Normalized execution time of SkyByte with different thread scheduling policies (lower is better).

to have equal or similar opportunities to issue memory requests to the SSD, even though these threads may trigger a context switch immediately after they are scheduled. This improves the SSD bandwidth utilization with multi-threading (see §VI-C). For some workloads (e.g., srad), context switching takes a considerable amount of time because context switches are repeatedly triggered by all threads when they are all waiting for flash accesses. The CFS policy may perform slightly worse in a few workloads, because it needs to enforce fair sharing for all the threads, which may cause the OS to select the threads that have just been scheduled away due to the context switch in SkyByte. But its impact on the end-to-end performance is trivial. Since CFS has become a standard scheduling policy in modern OSes like Linux, we employ it by default in SkyByte.

B. CXL-Aware SSD DRAM Management

As discussed in §II-C, the internal DRAM cache of SSDs can be suboptimal due to the mismatch of the access granularity between the CXL interface and the flash memory. A cacheline request would trigger a flash page access, which will cause a long delay, and waste precious SSD DRAM capacity.

To address these issues, we re-architect the SSD DRAM, as shown in Figure 11. SkyByte deploys a cache-line granular *double-buffered write log* to buffer all write requests from the host. All cacheline writes are directly appended to the log without flash access along the critical path. They are flushed to the flash memory later. Compared to a page-granular cache, the write log has two benefits: (1) caching at a finer granularity saves the precious SSD DRAM space when the write locality is bad, and (2) the write log provides a larger write coalescing window, which reduces the flash write traffic. We maintain a
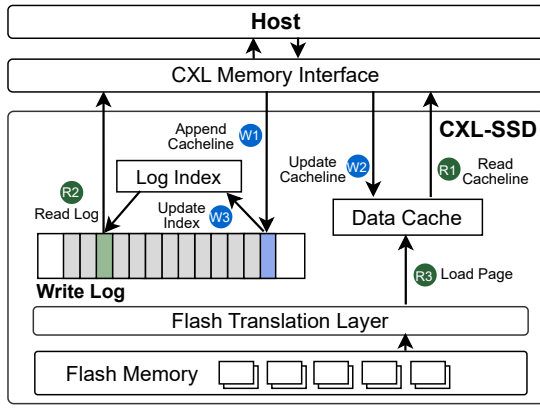
Fig. 11: The architecture of CXL-aware SSD DRAM.

Fig. 12: The two-level hash table structure for indexing the write log.

log indexing table to index the latest data in the write log array for read requests or log compaction.

For read operations, when flash access is inevitable upon an SSD DRAM miss, we still need to fetch the entire page. Thus, SkyByte caches the fetched page with a read-write data cache managed in page granular to exploit data locality. To maintain data consistency between the write log and data cache, we update both the write log and cache upon a write request, and during a read, we check the write log first. To speed up the requests, SkyByte performs parallel updates and parallel lookups on both sides. The write log and the data cache use logical addresses for indexing, as they are built on top of the Flash Translation Layer (FTL). We discuss them as follows.

Write log structure. Figure 12 shows the structure of the write log. The log records all written 64B cache lines within a circular buffer with head and tail pointers. To achieve a fast lookup of log entries, SkyByte uses the hash table for indexing, as it provides an amortized $O(1)$ lookup latency. However, a plain hash table will randomly distribute the stored entry, and it would require multiple lookups when finding all cache lines within the same page during compaction. To solve the problem, SkyByte breaks the indexing structure into two levels. At the first level, we employ a hash table indexed by the logical page address (LPA). Each valid entry points to a *second level hash table* that tracks all logged cache lines in this logical flash page with their offset within the page. The log offset of the tracked cache line in the log is indexed in the second-level table. We can easily find all updated cache lines in the same page by traversing the corresponding second-level hash table.

In the first-level hash table, each entry stores the 8B LPA and 8B second-level hash table pointer for each page. Since each page (4KB) has 64 cache lines, we only need 6 bits to index the page offset. The second-level hash table entry is 4B, which includes the 6-bit page offset and a 26-bit log offset. Consider a 64MB write log with 1M entries, if all second-level hash tables are initially allocated with full 64 entries, it requires 272MB of memory under the worst case when each page only contains a single dirty cache line. To reduce the memory footprint, we instead allocate small-sized
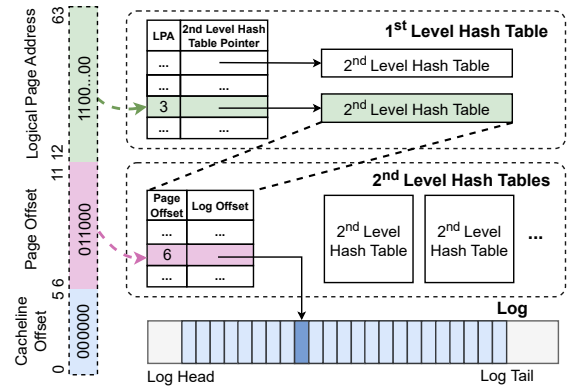
second-level hash tables and allow them to resize on demand. SkyByte initiates each second-level hash table with four entries (16B), and doubles the table size whenever the table load factor exceeds a threshold (0.75 by default). With resizing, under the worst case, our memory footprint only occupies up to 32MB (1M 16B first-level hash entries and 1M 16B second-level hash tables). Our experiments with real workload traces (§VI) show that the log index occupies 5.6MB of memory on average.

Read operation. We show the read/write operation in Figure 11. When a read request arrives, the SSD controller looks up both the data cache and the write log in parallel. If the requested data is cached, we directly read from the data cache and return to the host (R1). When the data cache misses but the write log holds the cache line, SkyByte will retrieve it from the write log (R2). When both write log and data cache miss, the entire page is fetched from the flash into the data cache, and SkyByte returns the target cache line (R3). The write log may contain the recently updated cache lines, and we need to keep the cached page up-to-date. After fetching the page to the cache, SkyByte performs a lookup on the first-level hash table. If it contains the entry of this page, we traverse its second-level table to merge all cache lines in the log to the fetched page.

Write operation. On a write request, SkyByte directly appends the written cache line at the tail of the write log (W1). An update to the data cache is issued in parallel if it contains the corresponding page (W2). SkyByte will also update the indexing table (W3). If the write updates an existing cache line in the write log, SkyByte updates the index table entry pointing to the newest log offset. SkyByte serves multiple read/write requests in parallel and leverages lock-free hash tables [55] and queues for synchronization.

Write log compaction. Figure 13 shows how SkyByte performs the log compaction. To reduce write traffic to the flash chips, SkyByte coalesces the writes during log compaction. Since we only track the newest data in the indexing table, the old updates will be dropped during the compaction. SkyByte scans the first level hash table to find all pages that need to be flushed (L1). For each page, if it is cached, we directly flush the cached page back to flash memory (L2). Otherwise, we load the missing page from the flash memory to a coalescing
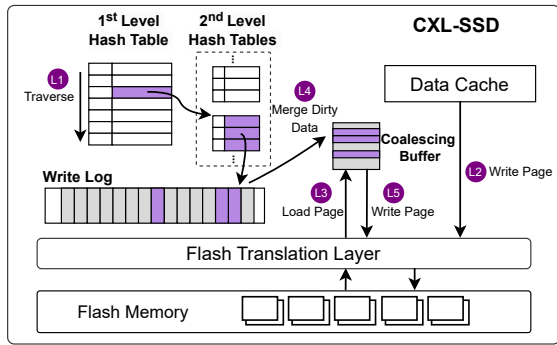
Fig. 13: Log compaction in CXL-SSD.

buffer (L3). SkyByte then traverse the second level table entries for dirty cache lines in this page and merge them with the loaded page (L4). The merged page is then written back to the flash (L5). When performing flash writes, SkyByte batches the pages in the write buffer, and distributes writes across multiple channels to exploit SSD channel parallelism.

SkyByte maintains a double-buffered log to avoid blocking incoming requests and performs the compaction in the background. When one log becomes full, SkyByte triggers the log compaction process and switches to a new log. During compaction, incoming write requests will be directed to the new log, following the normal write procedure. An incoming read request requires a parallel lookup in both the new log and the old log for the latest data. As the background compaction is not on the critical path of serving memory requests, it does not introduce much overhead, and a single compaction takes 146 μs on average. After compaction, we remove the indexing table and reclaim the memory used by the previous log.

C. Adaptive Page Migration

As the SSD DRAM size is limited, we use the host memory to expand SSD DRAM cache. SkyByte develops an adaptive page migration mechanism to migrate frequently accessed pages to the host. To decide which pages to migrate, SkyByte uses a similar policy developed in prior work [7], [8], [14], [19], [35], [48], [61]. The SSD controller tracks the access count of flash pages and selects pages whose access counts exceed a threshold as the migration candidates. SkyByte only migrate pages in the SSD DRAM cache, as it includes the candidate hot pages. After choosing the target page for migration, the SSD triggers the migration by sending a PCIe MSI-X interrupt with the SSD page address to the host. The host OS then allocates a physical page in the host DRAM memory with the default buddy allocator and copies the page content to the new page.

We need to ensure data consistency during page migrations. To achieve this, a simple approach is to leverage the OS techniques by setting the page under migration as not presented in PTE before the migration, and revising the page fault handler to resume the request after the migration. However, this causes high performance overheads. In SkyByte, we follow the prior approach [7] and track the migration progress with a Promotion Look-aside Buffer (PLB) in the root complex. The PLB has 64 entries, each entry (24B) records all ongoing migrations with

the addresses of source/destination pages (8B each), a bitmap for the migrated cache line (8B), and a valid bit. Therefore, a read request to a page under promotion can be served from the SSD DRAM. For writes, if the migrated bit has been set for the requested cache line, the request is forwarded to the most recent copy of that cache line in the host DRAM.

After data migration, the host OS modifies the page table entry (PTE) to map the original virtual address to the new host DRAM page, with the corresponding TLB entry also updated. The host OS will acknowledge the migration request, and the SSD removes the page from the data cache and invalidates the write log index by setting the corresponding entry as `NULL`.

Since the host DRAM space is limited, SkyByte also enables the host to evict pages back to SSD for free space. SkyByte leverages the existing page reclamation policy [3] in Linux to select the page for eviction, finding a relatively "cold" page tracked by the active/inactive list [6]. We then allocate a new page in the CXL memory space and perform the page copy. The host OS will update the corresponding page table entry to point at the SSD page and update the TLB.

IV. DISCUSSION

Data persistence support. By default, SkyByte assumes the user application does not require data persistence, so it can transparently promote pages to the host DRAM for performance improvement. To support data persistence, SkyByte offers the option for users to pin memory pages in the CXL-SSD, such that these pages will not be promoted to the host. Programmers can use `clwb` instruction to ensure a cacheline has reached the battery-backed SSD DRAM.

Support for NUMA architecture. SkyByte can work with a multi-socket NUMA machine. The CXL-SSD appears to the system as a "CPU-less" NUMA node. All processors treat the CXL-SSD as non-local memory. The CXL-SSD is attached to the PCIe slot of one CPU socket, referred to as the "home node". Accesses from other NUMA nodes to the CXL-SSD may experience slightly higher latency compared to accesses from the home node, but since the inter-socket latency is much smaller (less than 100 ns) than the flash latency (μs-level), SkyByte uses the same context switch threshold for all NUMA nodes. Since the SSD controller is unaware of which NUMA nodes have accessed each page, when a page migration is triggered, the page is migrated to the home node first. If the home node has no free memory, the page will be migrated to the NUMA node with the most free memory space. After that, if the page is needed by other nodes, the NUMA balancing mechanism of the OS is responsible for further migrations.

Support for multiple page sizes. The adaptive page migration of SkyByte can support multiple page sizes (e.g., 2MB huge pages). When the host OS receives the MSI-X interrupt from the SSD with a 4KB-page address, it first checks whether this address belongs to a huge page. If it does, the host OS allocates a physical huge page in the host DRAM, and migrates the entire page by copying all 4KB data chunks from the SSD. Once the migration finishes, the host sends a custom NVMe

command to notify the SSD to remove all corresponding 4KB chunks from its internal DRAM caches.

To migrate a huge page, the PLB needs to track all cachelines in the huge page. However, tracking all 32,768 cachelines in a 2MB page requires a 4KB bitmap per PLB entry. To reduce the hardware cost of PLB, we extend the original PLB into a two-level structure. The first-level entry contains a 64B bitmap that indicates whether a 4KB chunk in a 2MB page has been migrated. The second-level entry contains an 8B bitmap to track which cachelines in a 4KB chunk have been migrated. The PLB migrates the huge page chunk-by-chunk, and it only needs one first-level entry to track the 2MB page and one second-level entry to track the current 4KB chunk under migration.

V. IMPLEMENTATION

SkyByte simulation framework. We implement SkyByte with a cycle-accurate simulator based on MacSim [41] and SimpleSSD [21]. MacSim replays multi-threaded instruction traces captured by Intel's PIN tool [30] on multiple simulated CPU cores. To implement SkyByte's coordinated context switch mechanism, we extend it to support task scheduling by selecting which set of instructions to execute on the simulated cores based on the scheduling policy discussed in §III-A. We measured the context switch overhead with an Intel E5 CPU under real systems, and we set our experiment timing model accordingly (see Table II). We also simulate the side effects of context switching, including cache contention or branch mispredictions. We implement the PLB (§III-C) in MacSim and enable the CPU to issue CXL memory requests to the SSD. To simulate the side effects of page migration, we also modified MacSim to perform a TLB shootdown for all cores when a page finishes migration. To simulate the CXL.mem interface and flash accesses in SSDs, we extend MacSim's memory controller logic to redirect all CXL.mem requests to the SSD simulator. The SSD simulator simulates the SSD firmware, including the core FTL functions (e.g., address translation and GC), SkyByte's write log and data cache in the SSD DRAM, and flash accesses.

FPGA SoC prototype. We prototype the write log and SSD DRAM cache (§III-B) on a Xilinx Zynq UltraScale+ ZU3EG MPSoC board. The board features a quad-core ARM Cortex A53 processor commonly deployed in SSD controllers [1], an LPDDR4 memory, and programmable logic resources. We implement critical-path operations such as indexing the write log and data cache on FPGA. Other off-critical-path tasks like log compaction and garbage collection are managed by the ARM cores. We use a red-black tree to index the page-granular data cache. The average lookup latency is 72 ns for a 64 MB write log and 49 ns for a 512 MB data cache. The latency grows slightly (by less than 10 ns) with a larger log size. Our prototype achieves a peak throughput of 11.93/9.37 GB/s for cacheline reads/writes. We verified the performance model used in our simulator with these measurements.

VI. EVALUATION

Our evaluation shows that: (1) SkyByte outperforms state-of-the-art CXL-SSD designs by $6.11\times$ on average (§VI-B);

TABLE I: Benchmarks used in our experiments.

Category	Suite	Name	Memory Footprint	Write Ratio	LLC MPKI
Graph Processing	Rodinia [17]	bfs-dense	9.13GB	25%	122.9
	GAP [13]	bc	8.18GB	11%	39.4
HPC	Splashv3 [51]	radix	9.60GB	29%	7.1
Image Processing	Rodinia [17]	srad	8.16GB	24%	7.5
Database	WHISPER [45]	ycsb	9.61GB	5.0%	92.2
		tpcc	15.77GB	36%	1.0
Machine Learning	DLRM [46]	dlrm	12.35GB	32%	5.1

TABLE II: Parameters defined in our CXL-SSD simulator.

CPU	Cores	8 cores, 4.0 GHz, 256 ROB entries per core
	L1 I/D Cache	32/32 KB, 8/8 ways per core, 8 MSHRs
	L2 Cache	512 KB, 32 ways per core, 128 MSHRs
	L3 Cache	16 MB, 16 ways, shared by all cores, 1024 MSHRs
	DRAM	DDR5 4800 MHz, 8 channels, 36-38-38 Max. Total Size of Promoted Pages: 2 GB
SSD	Interface	CXL over PCIe 5.0 x4 (16 GB/s, 40 ns protocol latency)
	Organization	16 channels, 8 chips/channel, 8 dies/chip, 1 plane/die, 128 blocks/plane, 256 pages/block, 4KB page (Total: 128 GB)
	Flash Latency	Read (t_R): 3 μs, Program (t_{Prog}): 100 μs, Erase (t_{BERS}): 1000 μs
	DRAM	LPDDR4 3200 MHz, 2 channels, 16-18-18 Data Cache Size: 512 MB, 2048 MSHRs
	GC Policy	Threshold: 80%, # of Blocks to Erase: 19660
Context Switch Overhead: 2 μs; Context Switch Trigger Threshold: 2 μs Write Log: 64 MB; Data Cache: 448 MB		

(2) SkyByte scales the performance with more threads by exploiting the context-switching opportunities to hide SSD access latency (§VI-C); (3) SkyByte reduces the average memory access time by $14.19\times$ and the flash write traffic by $23.08\times$ over the state-of-the-art CXL-SSD design (§VI-D); (4) SkyByte's write log design uses the SSD DRAM capacity efficiently (§VI-E); (5)SkyByte benefits CXL-SSDs with various internal DRAM sizes and flash chip latencies (§VI-F and §VI-G).

A. Experiment Setup

Workloads. We evaluate representative multi-threaded data-intensive workloads of various domains, including scientific computing, machine learning training, and database queries, as shown in Table I. All benchmarks are configured to have a total memory footprint of at least 8GB. The write ratio of the workloads ranges from 5% to 36%, and the LLC misses per kilo-instructions (MPKI) range from 1.0 to 122.9. We run the workloads on a server with Intel Xeon processors and capture the instruction traces for each thread using Intel's PIN tool [30]. The traces for ycsb and tpcc are collected using the in-memory database nstore [10], and we use workload B in ycsb. For each workload, we capture the traces for at least 100 million instructions per thread, and replay them in our simulator. In our experiments, all data are initially stored in CXL-SSD. We use the traces to warm up the simulator, including the CPU caches, the host memory, the SSD DRAM cache, and the write log. Page tables and program binaries are in the host memory. We precondition the SSD to ensure garbage collections will be triggered.

System configurations. Table II lists the simulator parameters. Following Samsung's recent 2TB CXL-SSD prototype with 16GB DRAM [24], [50], we scale down the configuration
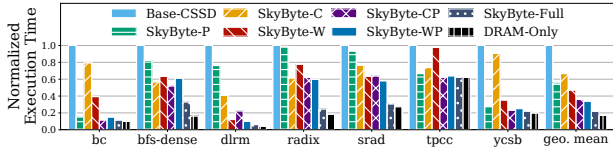
Fig. 14: Normalized execution time of SkyByte variants over Base-CSSD (lower is better).
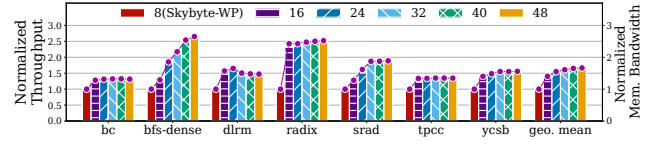

Fig. 15: Throughput (bars) and SSD bandwidth utilization (lines) of SkyByte-Full as we increase the number of threads (normalized to SkyByte-WP with 8 threads).

with the same flash-to-DRAM capacity ratio (i.e., 128GB flash with 512MB DRAM cache, excluding the FTL mapping table cache), as it is impractical to simulate a TB-scale SSD at cache line granularity. We use the NAND flash latency of Samsung Z-SSD SZ985, which uses ultra-low-latency (ULL) flash chips. Since modern servers usually have 64GB or larger DRAM (4× the DRAM size of Samsung's CXL-SSD), we configure the maximum host DRAM size for storing migrated pages as 2GB (4× our simulated SSD DRAM size). We also conduct sensitivity analysis on SSD DRAM cache size and flash latency.

We conduct an ablation study by testing SkyByte with subsets of design components enabled. When the coordinated context switch mechanism is enabled, we run 24 threads on 8 cores for all workloads. We also conduct sensitivity analysis by varying the number of threads. For other cases, we run 8 threads on 8 cores since more threads will not improve the performance. We run each workload multiple times with different numbers of threads. In each run, we collect a separate trace for each thread. We ensure that all traces represent the same section of the program. We compare the following designs:

- **Base-CSSD**: the state-of-the-art CXL-based SSD device that incorporates all optimizations developed in recent works [32], [62], including prefetching from flash to SSD DRAM, optimized cache replacement policy, and MSHRs in the SSD controller for tracking accesses to the flash chips.
- **SkyByte-C**: Base-CSSD with SkyByte's coordinated context switch mechanism.
- **SkyByte-P**: Base-CSSD with adaptive page migration.
- **SkyByte-W**: Base-CSSD with SkyByte's CXL-aware SSD DRAM management (the write log and data cache).
- **SkyByte-CP**: Base-CSSD with both SkyByte's coordinated context switch mechanism and adaptive page migration.
- **SkyByte-WP**: Base-CSSD with both page migration and CXL-aware SSD DRAM management.
- **SkyByte-Full**: the complete version of SkyByte that enables context switches upon SkyByte-WP.
- **DRAM-Only**: the ideal case assuming we run the workload with infinite host DRAM.

B. Overall Performance Improvement

Figure 14 shows the performance of different SkyByte variants normalized to Base-CSSD. Overall, SkyByte-Full outperforms Base-CSSD by up to 16.35× (6.11× on average).

SkyByte-P's performance reflects the benefit from page promotions (§III-C). It outperforms Base-CSSD by 1.84× on average by using host memory to expand the SSD DRAM cache size. Workloads with better locality (e.g., bc, tpcc,

and ycsb) benefit more from page promotions, while other workloads (e.g., radix and srad) benefit less.

SkyByte-W's performance reflects the benefit of the write log (§III-B). It outperforms Base-CSSD by 2.16× on average. By buffering and coalescing write requests, SkyByte-W significantly reduces the writeback traffic to the flash chips and hence reduces the overall latency of flash access requests. The benefit is especially obvious for temporally sparse workloads like bc and dlrm, which are otherwise frequently bottlenecked by write misses in Base-CSSD. Workloads that have many sparse writes (e.g., srad) benefit more from SkyByte-W, while workloads that are sensitive to the DRAM cache size and have relatively fewer sparse writes (e.g., tpcc) benefit more from SkyByte-P. When combining both optimizations, SkyByte-WP outperforms Base-CSSD by 2.95× on average.

SkyByte-CP leverages host DRAM as a cache and uses context switches to hide I/O latency. It outperforms Base-CSSD by 2.79×. SkyByte-Full further outperforms SkyByte-CP by 1.64× on average, as it also utilizes the write log to address the granularity mismatch issue of CXL-SSD.

SkyByte-Full outperforms SkyByte-WP by 1.55× on average and Base-CSSD by 1.61–16.35×. While SkyByte-WP already represents a well-optimized CXL-SSD device, its ideal performance is still bottlenecked by the long flash accesses. SkyByte-Full enables opportunistic context switches and leverages multi-threading to further hide long flash delays. By comparing the performance improvement of SkyByte-C over Base-CSSD and that of SkyByte-Full over SkyByte-WP (both are 1.49× on average), we observe that the benefit of context switching is orthogonal to that of write log and page migration, except for tpcc, where the benefit of page migration dominates.
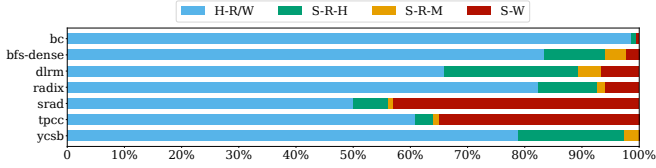
Compared to infinite host DRAM (DRAM-Only), SkyByte-Full achieves 75% of the ideal performance with CXL-SSD. Given the unit cost of DDR5 DRAM [5] and ULL SSD [4] is $4.28/GB and $0.27/GB respectively based on their market price in summer 2024, SkyByte-Full costs 15.9× less than the DRAM-only setup and improves cost-effectiveness by 11.8×.

C. Benefit of Context Switch with Varying Number of Threads

We analyze the throughput improvements of the coordinated context switch mechanism (§III-A) by varying the number of threads with SkyByte-Full on 8 CPU cores, as shown in Figure 15. The throughput is strongly correlated to the memory bandwidth utilization improvement with multi-threading. The potential of such improvement depends on the flash read latency (Table III) and the percentage of flash reads over all memory accesses (Figure 16). If most accesses are absorbed by the host

TABLE III: Average flash read latency of SkyByte-WP.

Workload	bc	bfs-dense	dlrm	radix	srad	tpcc	ycsb
Latency (μs)	3.5	25.7	3.4	4.9	22.5	19.6	3.3

Fig. 16: Breakdown of all memory requests of SkyByte[1]. H-R/W: host DRAM read/write. S-R-H: CXL-SSD DRAM read hit. S-R-M: CXL-SSD DRAM read miss. S-W: CXL-SSD write.

(a) Average memory access time (normalized to Base-CSSD).

(b) Average memory access time breakdown.

Fig. 17: Average memory access time (AMAT) in SkyByte. (a) compares the AMAT across different designs. (b) shows the percentage breakdown of different AMAT components.

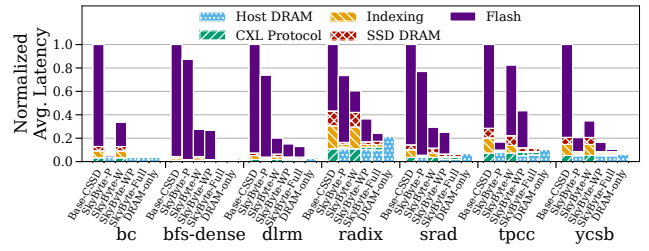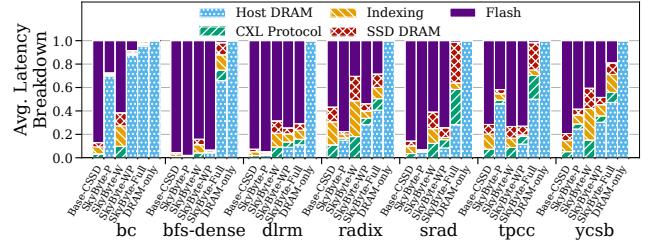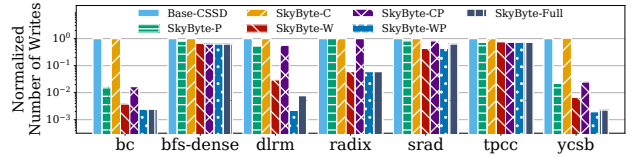**Fig. 18:** Write traffic of SkyByte to flash chips.

and SSD DRAM, context switching among multiple threads has limited benefits. Otherwise, with more flash accesses, the throughput scales linearly with the number of threads because of higher SSD bandwidth utilization. The benefit diminishes once the context switch overhead exceeds the flash read latency.

Table III and Figure 16 shows the average latency and the percentage of flash reads for SkyByte-WP. Workloads that suffer from more flash accesses and higher average flash read latency (e.g., `srad`, `bfs-dense`) benefit more from more threads. If the workload frequently triggers log compaction, the long flash writes will interfere with the flash reads, hence increasing the read latency. In this case, more threads can better hide the long latency. On the other hand, if the average flash read latency is already close to the context switch latency (e.g., for `bc` and `dlrm` in Table III), having two threads per core is sufficient to achieve optimal throughput. For some workloads (e.g., `dlrm`), the throughput drops when the context switch overhead becomes high with too many threads. With more threads, the latency of accessing flash chips may increase due to queuing delays. However, the end-to-end performance is improved, since the SSD bandwidth utilization is higher.

D. Benefit of CXL-Aware SSD DRAM Management

Average memory access time (AMAT). We break down the AMAT of Base-CSSD and SkyByte variants in Figure 17. To analyze the end-to-end memory latency observed by the host CPU, we consider the accesses to the promoted pages in the host DRAM as "host DRAM hits", while all other accesses go to the CXL-SSD and will suffer the CXL protocol latency (40 ns, see Table II). For all CXL-SSD accesses, we classify them as "SSD DRAM hits" (hits in write log or data cache) or "SSD DRAM misses" (which suffer the flash access latency). The latency of a CXL-SSD access consists of (1) the SSD DRAM cache indexing time (72 ns for the write log, 49 ns for the data cache, see §V), (2) the SSD DRAM access time (based on the timing in Table II), and (3) the flash chip latency (see Table II) in case of an SSD DRAM miss. Together, we calculate AMAT by modeling host DRAM, SSD DRAM, and flash chips as a three-level memory hierarchy, where access to SSD DRAM

[1]As we focus on the memory requests served by the CXL-SSD, we do not distinguish host reads and writes. We also do not distinguish CXL-SSD write hits and misses, as all CXL-SSD writes will append to the write log.

will bypass host DRAM. For SkyByte-Full, a memory access triggering a context switch is excluded from calculating AMAT since this instruction is squashed. The replayed instruction that eventually retires is included in AMAT.

With SkyByte-W, the write log greatly reduces the flash access latency for three reasons. First, writes are buffered in SSD DRAM without expensive flash accesses on the critical path. Second, by coalescing writes to the flash, SkyByte-W reduces the number of flash writes, which leads to less interference to flash reads and triggers GC less frequently. Third, by buffering the writes at cacheline granularity, SkyByte-W caches much fewer unused cachelines. This improves the efficiency of the SSD DRAM cache. With SkyByte-P, the page migration reduces the number of SSD accesses, and more memory requests are served by the faster host DRAM. With both optimizations, SkyByte-WP improves the overall AMAT.

With coordinated context switch, SkyByte-Full achieves better AMAT than SkyByte-WP, because the flash latency can be hidden from applications. Although the AMAT of SkyByte-Full is still $1.39\times$ that of the ideal DRAM-Only case on average, the end-to-end performance degradation is only $1.33\times$ on average (see Figure 14). This is because SkyByte-Full exploits the parallelism from more threads than DRAM-Only (24 vs. 8 threads on 8 CPU cores) to tolerate higher AMAT.

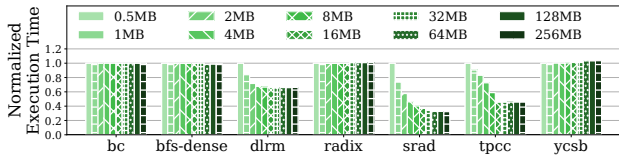**Flash write traffic.** Figure 18 shows the flash write traffic.

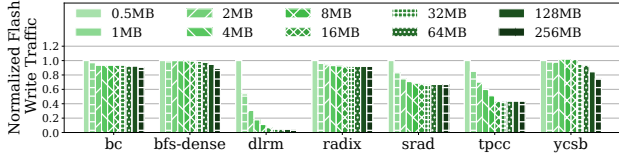**Fig. 19:** SkyByte performance with various write log sizes.

Fig. 20: Flash write traffic in SkyByte with various write log sizes.

Compared to Base-CSSD, both SkyByte-P and SkyByte-W reduce the flash write traffic by coalescing writes in the host or SSD DRAM. However, SkyByte-W is more effective because it enables a larger coalescing window with the cacheline-granular write log. The write log utilizes the limited DRAM space more efficiently, so it can buffer more writes in the limited SSD DRAM. SkyByte-WP achieves the benefits of both designs.

The context switch mechanism may slightly increase the flash write traffic (e.g., SkyByte-Full v.s. SkyByte-WP, and SkyByte-CP v.s. SkyByte-P). This is because context switches enable multiple threads to access the SSD simultaneously, increasing the contention in the SSD DRAM. As a result, the log compaction is triggered more frequently. This overhead is acceptable given its performance benefits.

E. Impact of Varying Write Log Size

We study the performance impact of varying write log size in Figure 19. We keep the total size of the write log and data cache in the SSD DRAM fixed (i.e., 512MB). The impact of the write log size mainly depends on the flash write traffic that can be reduced during log compaction, as shown in Figure 20. Workloads with more CXL-SSD writes (see Figure 16) and better temporal write locality will be more sensitive to write log size, as a larger write log can coalesce more flash writes (e.g., srad and tpcc). For most workloads, a small write log (e.g., no more than 64 MB, or 1/8 of the total SSD DRAM size) already provides a sufficiently large write coalescing window.

F. Impact of Varying SSD DRAM Size

To further understand the efficiency of the SSD DRAM management in SkyByte, we vary the SSD DRAM data cache size. In practice, the host DRAM size should scale with a larger SSD DRAM. Hence, we keep the ratio between the maximum size of the promoted pages in the host DRAM and the SSD DRAM cache size the same as the default (i.e., 4:1) in Table II. We also keep the ratio between the size of the write log and the data cache to 1:7 for a fair comparison.

Figure 21 shows the performance of SkyByte variants with various SSD DRAM cache sizes. In all cases, SkyByte-Full is better than all other baselines. The major benefit comes from the efficiency of the cacheline-granular write log, as it enables

TABLE IV: The NAND flash parameters used in our evaluation.

NAND Type	SSD Device	Read Time	Program Time	Erase Time
ULL	Samsung Z-NAND SSD [52]	3 μs	100 μs	1000 μs
ULL2	Toshiba XL-Flash [33]	4 μs	75 μs	850 μs
SLC	[62]	25 μs	200 μs	1500 μs
MLC	[62]	50 μs	600 μs	3000 μs

a larger effective cache size compared to a page-granular cache. SkyByte helps reduce the cost of CXL-SSDs, as it can achieve similar or better performance with a smaller SSD DRAM compared to Base-CSSD with a much larger SSD DRAM.

G. Impact of Varying Flash Latency

Figure 22 shows SkyByte's performance with different flash chips, including fast ULL flash chips and slower SLC/MLC chips [62] (see Table IV). With a higher flash latency, both SkyByte-WP and SkyByte-Full achieve higher improvement over SkyByte-P. As the benefits of write log and context switching come from the ability to hide flash latency, their benefits are more obvious with a higher flash latency. SkyByte-Full can utilize more threads to hide the flash latency and improve the application performance. As long as the SSD bandwidth is enough, SkyByte-Full can scale the performance using more threads until the context switch latency dominates the execution, which aligns with our findings in §VI-C. SkyByte demonstrates that it is promising to use slower yet cheaper commodity flash chips to build CXL-SSDs for parallelizable applications, as they can achieve similar performance with SkyByte compared to the cutting-edge Z-NAND flash chips.

H. Comparison with Alternative Page Migration Mechanisms

Prior studies have proposed various page management mechanisms for tiered memory systems. While SkyByte employs an adaptive page migration mechanism (see §III-C) by default, in this section, we study the effect of applying alternative page management policies in SkyByte. We introduce the following designs: (1) **SkyByte-CT** and (2) **SkyByte-WCT**. They replace SkyByte's adaptive page migration with the software-based page migration mechanism of TPP [43]. TPP extends Linux's NUMA balancing and uses periodic sampling and Linux's LRU list to identify hot pages to promote from CXL memory to the host DRAM. SkyByte-WCT enables the write log in SSD DRAM, while SkyByte-CT does not. (3) **AstriFlash-CXL** applies AstriFlash [23] on Base-CSSD. It employs the host DRAM as a hardware-managed set-associative cache of SSD at 4KB page granularity. It also uses user-level thread switches triggered by host DRAM misses to hide the SSD I/O latency. Since AstriFlash always accesses the SSD at page granularity, we do not integrate it with SkyByte's write log design.

Compared to SkyByte-C, all of SkyByte-CT, SkyByte-CP, and AstriFlash-CXL improve performance by caching hot pages in the host DRAM (see Figure 23). SkyByte-CT performs slightly worse than SkyByte-CP on average because TPP uses periodic sampling to estimate page hotness, which is less accurate than the per-page tracking in SkyByte. SkyByte-CP outperforms AstriFlash-CXL by up to 1.21× (1.09× on
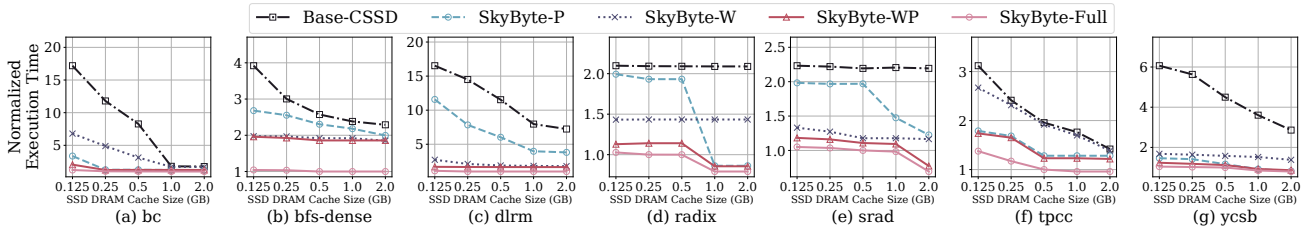
Fig. 21: Performance of SkyByte with varying SSD DRAM cache size (normalized to SkyByte-Full with default SSD DRAM cache size).
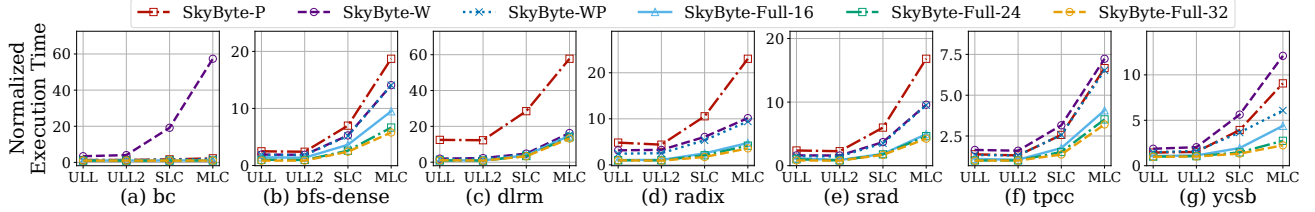

Fig. 22: SkyByte performance with varying flash latency. We vary the thread count in SkyByte-Full (e.g. SkyByte-Full-16 uses 16 threads).
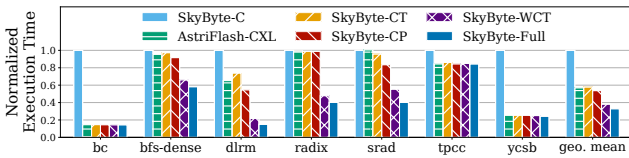

Fig. 23: Execution time of SkyByte with different page migration mechanisms (normalized to SkyByte-C).

average), as SkyByte utilizes the host DRAM more efficiently since it only promotes hot pages and effectively uses the host DRAM as a fully associative cache. In contrast, AstriFlash relies on on-demand paging and manages the host DRAM as a set-associative cache. SkyByte-WCT outperforms SkyByte-CT by 1.10× on average, demonstrating that SkyByte's CXL-aware SSD DRAM management design can be applied to improve the performance of TPP with CXL-SSD. SkyByte-Full further improves over SkyByte-WCT by 1.16× on average.

VII. RELATED WORK

CXL-based Memory Architecture. The recent development of CXL technology [18] motivates researchers to rethink the design and deployment of existing memory techniques [9], [31], [37], [43], [59], [62], [63]. For instance, they investigated the impact of CXL on disaggregated memory [9], [22], [37], and identified new opportunities to facilitate the deployment of tiered memory [43], [64]. CXL has also been applied to SSDs [62]. We target such a new architecture, investigate its bottlenecks, and propose a holistic approach to make CXL-based SSDs truly usable with software/hardware co-design.

Memory-Semantic SSDs. Using SSDs to expand main memory capacity has been explored in prior studies [16], [27], [53], [56]. However, they still treated SSDs as block devices and relied on OS paging mechanisms to manage the data movement between the SSD and host memory. A few studies exploited the byte-accessibility of SSDs [7], [11] and examined its impact on software systems such as ByteFS [39]. SkyByte advances the CXL-SSD with new OS and storage architecture supports.

New and Emerging Memory Architecture. To overcome the memory wall, alternative memory technologies such as non-volatile memories have been developed [15], [28], [34], [36]. Compared to these memory technologies, memory-semantic SSDs provide the generic memory interface while offering scalable memory capacity with much lower cost. To enhance SSD performance, prior studies have developed various architecture-level [27], [57] and device-level optimization techniques [40], [52]. SkyByte is compatible with these works, with a focus on addressing the performance challenges in the context of CXL. Prior studies have proposed informing memory operations that enable the software to proactively react to specific memory events such as cache misses [25]. They provided new instruction primitives and hardware support for informing memory events. Inspired by these studies, SkyByte uses the SSD controller to inform the host CPU when a long flash access will happen. AstriFlash [23] uses the host DRAM as a hardware-managed cache of the SSD, and hides the flash latency using user-level thread switches triggered by host DRAM misses. However, it still treats the SSD as a black box and manages it at page granularity. SkyByte co-designs the host and the SSD to address the challenges of CXL-SSDs.

VIII. CONCLUSION

We investigate the performance bottlenecks of CXL-based SSDs and their impact on the application performance. We employ a holistic approach to develop SkyByte by co-designing the host OS and SSD, with coordinated context switch and CXL-aware SSD DRAM management. We show that SkyByte outperforms current CXL-based SSDs by 6.11× on average.

REFERENCES

[1] "Arm storage solution for ssd controllers," https://armkeil.blob.core.windows.net/developer/Files/pdf/solution-brief/arm-storage-solution-for-ssd-solutions-brief.pdf.

[2] "Cfs scheduler - the linux kernel documentation." [Online]. Available: https://docs.kernel.org/scheduler/sched-design-CFS.html

[3] "Page reclaim – the linux kernel documentation." [Online]. Available: https://www.kernel.org/doc/gorman/html/understand/understand013.html

[4] "Pm1733/pm1735 — enterprise ssd — samsung semiconductor usa." [Online]. Available: https://semiconductor.samsung.com/us/ssd/enterprise-ssd/pm1733-pm1735/

[5] "Userbenchmark: Samsung-m321r8ga0bb0-cqkzj-2x315gb." [Online]. Available: https://ram.userbenchmark.com/SpeedTest/2269793/Samsung-M321R8GA0BB0-CQKZJ-2x315GB

[6] "workingset.c - mm/workingset.c - Linux source code (v6.10) - Bootlin." [Online]. Available: https://elixir.bootlin.com/linux/v6.10/source/mm/workingset.c

[7] A. Abulila, V. S. Mailthoday, Z. Qureshi, J. Huang, N. S. Kim, J. jun Xiong, and W. mei Hwu, "FlatFlash: Exploiting the Byte-Accessibility of SSDs within A Unified Memory-Storage Hierarchy," in *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, Providence, RI, 2019.

[8] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent Page Management for Two-tiered Main Memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, Xi'an, China, 2017, pp. 631–644.

[9] M. K. Aguilera, E. Amaro, N. Amit, E. Hunhoff, A. Yelam, and G. Zellweger, "Memory disaggregation: why now and what are the challenges," *SIGOPS Operating System Review*, vol. 57, no. 1, p. 38–46, June 2023.

[10] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 707–722. [Online]. Available: https://doi.org/10.1145/2723372.2749441

[11] D.-H. Bae, I. Jo, Y. A. Choi, J.-Y. Hwang, S. Cho, D.-G. Lee, and J. Jeong, "2B-SSD: The Case for Dual, Byte- and Block-Addressable Solid-State Drives," in *Proceedings of the 45Th Annual International Symposium on Computer Architecture*, ser. ISCA '18, Los Angeles, CA, 2018, pp. 425–438.

[12] C.-H. C. M. M. H. S. C. based SSD for the Memory-centric Computing Era, https://semiconductor.samsung.com/us/news-events/tech-blog/webinar-memory-semantic-ssd/.

[13] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 2017.

[14] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management*. Oreilly & Associates Inc, 2005.

[15] M. Cai, C. C. Coats, and J. Huang, "Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory," in *Proceedings of ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*, 2020, pp. 584–596.

[16] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV, Washington, DC, 2009, pp. 217–228.

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[18] Compute Express Link Specification 3.0, https://computeexpresslink.org/cxl-specification/.

[19] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10, Washington, DC, 2010, pp. 1–11.

[20] S. K. Gonugondla, M. Kang, Y. Kim, M. Helm, S. Eilert, and N. Shanbhag, "Energy-efficient deep in-memory architecture for nand flash memories," in *Proceedings of 2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018.

[21] 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 *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 469–481.

[22] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access, High-Performance memory disaggregation with DirectCXL," in *Proceedings of 2022 USENIX Annual Technical Conference (USENIX ATC'22)*, Carlsbad, CA, Jul. 2022, pp. 287–294.

[23] S. Gupta, Y. Oh, L. Yan, M. Sutherland, A. Bhattacharjee, B. Falsafi, and P. Hsu, "Astriflash a flash-based system for online services," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 81–93.

[24] J. Handy, "Understand how the cxl ssd can aid performance," https://www.techtarget.com/searchstorage/feature/Understand-how-the-CXL-SSD-can-aid-performance, 2023.

[25] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith, "Informing memory operations: providing memory performance feedback in modern processors," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ser. ISCA '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 260–270. [Online]. Available: https://doi.org/10.1145/232973.233000

[26] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi, "FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs," in *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST'17)*, Santa clara, CA, 2017.

[27] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan, "Unified Address Translation for Memory-mapped SSDs with FlashMap," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, Portland, OR, 2015, pp. 580–591.

[28] J. Huang, K. Schwan, and M. K. Qureshi, "NVRAM-award Logging in Transaction Systems," in *Proceedings of the 41th International Conference on Very Large Data Bases (VLDB'15)*, Kohala Coast, HI, 2015.

[29] Intel, "Cpu metrics reference," https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-0/cpu-metrics-reference.html.

[30] Intel Corporation, "Pin - A Dynamic Binary Instrumentation Tool."

[31] J. Jang, H. Choi, H. Bae, S. Lee, M. Kwon, and M. Jung, "CXL-ANNS: Software-Hardware collaborative memory disaggregation and computation for Billion-Scale approximate nearest neighbor search," in *Proceedings of 2023 USENIX Annual Technical Conference (USENIX ATC 23)*, Boston, MA, Jul. 2023.

[32] 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'22)*, Virtual Event, 2022.

[33] Kioxia, "Xl-flash storage class memory (scm)," https://americas.kioxia.com/en-us/business/memory/xlflash.html.

[34] A. Kokolis, A. Psistakis, B. Reidys, J. Huang, and J. Torrellas, "Distributed data persistency," in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'21)*, Virtual Event, Greece, 2021.

[35] J. Lee and H. Kim, "TAP: A TLP-aware Cache Management Policy for a CPU-GPU Heterogeneous Architecture," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA'12)*, Washington, DC, 2012, pp. 1–12.

[36] D. Li, B. Reidys, J. Sun, T. Shull, J. Torrellas, and J. Huang, "Uniheap: managing persistent objects across managed runtimes for non-volatile memory," in *Proceedings of the 14th ACM International Conference on Systems and Storage (SYSTOR'21)*, Haifa, Israel, 2021.

[37] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*, Vancouver, BC, Canada, 2023.

[38] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, Feb. 2018, pp. 83–90. [Online]. Available: https://www.usenix.org/conference/fast18/presentation/li

[39] 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'25)*, Rotterdam, Netherlands, 2025.

[40] C.-Y. Liu, J. B. Kotra, M. Jung, M. T. Kandemir, and C. R. Das, "Soml read: Rethinking the read operation granularity of 3d nand ssds," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, Providence, RI, USA, 2019.

[41] MacSim: A Cycle-Level, Heterogeneous Architecture Simulator, http://comparch.gatech.edu/hparch/macsim.html.

[42] V. S. Mailthoday, Z. Qureshi, W. Liang, Z. Feng, S. G. de Gonzalo, Y. Li, H. Franke, J. Xiong, J. Huang, and W. mei Hwu, "DeepStore: In-Storage Acceleration for Intelligent Queries," in *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*, Columbus, OH, Oct. 2019.

[43] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "Tpp: Transparent page placement for cxl-enabled tiered-memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*, Vancouver, BC, Canada, 2023.

[44] R. Micheloni and L. Crippa, *Solid-State-Drives (SSDs) Modeling*. Springer, 2017.

[45] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 13 –148. [Online]. Available: https://doi.org/10.1145/3037697.3037730

[46] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep learning recommendation model for personalization and recommendation systems," *CoRR*, vol. abs/1906.00091, 2019. [Online]. Available: https://arxiv.org/abs/1906.00091

[47] M. Nemirovsky and D. Tullsen, *Multithreading Architecture*, ser. Synthesis Lectures on Computer Architecture. Springer International Publishing, 2022. [Online]. Available: https://books.google.com/books?id=SYZyEAAAQBAJ

[48] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11, Tucson, AZ, 2011, pp. 85–95.

[49] B. Reidys, J. Sun, A. Badam, S. Noghabi, and J. Huang, "BlockFlex: Enabling storage harvesting with Software-Defined flash in modern cloud platforms," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, Carlsbad, CA, Jul. 2022.

[50] C. Robinson, "Samsung memory-semantic cxl ssd at fms 2022 powered by amd-xilinx," https://www.servethehome.com/samsung-memory-semantic-cxl-ssd-at-fms-2022-powered-by-amd-xilinx/, 2022.

[51] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 101–111.

[52] Samsung Z-NAND, https://www.samsung.com/semiconductor/ssd/z-ssd/.

[53] M. Saxena and M. M. Swift, "FlashVM: Virtual Memory Management on Flash," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10, Boston, MA, 2010, pp. 187–200.

[54] S. Semiconductor, "Cmm-h (cxl memory module - hybrid): Samsung's cxl-based ssd for the memory-centric computing era," https://semiconductor.samsung.com/us/news-events/tech-blog/webinar-memory-semantic-ssd/?CID=afl-ecomm-rkt-cha-040122-url_Cashback+on+Bing+for+Edge+browser&utm_source=url_Cashback+on+Bing+for+Edge+browser&utm_medium=affiliate&utm_campaign=1&utm_content=3829940&rktevent=Cashback+on+Bing+for+Edge+browser__jZHTpnCvx8-u7RSEurkbuqr0ZUBaUdkiw&ranMID=47773&ranEAID=%2FjZHTpnCvx8&ranSiteID=_jZHTpnCvx8-u7RSEurkbuqr0ZUBaUdkiw, 2023.

[55] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," *J. ACM*, vol. 53, no. 3, p. 379–405, may 2006. [Online]. Available: https://doi.org/10.1145/1147954.1147958

[56] N. Y. Song, Y. Son, H. Han, and H. Y. Yeom, "Efficient Memory-Mapped I/O on Fast Storage Device," *Trans. Storage*, vol. 12, no. 4, pp. 19:1–19:27, May 2016.

[57] J. Sun, S. Li, Y. Sun, C. Sun, D. Vucinic, and J. Huang, "Leaftl: A learning-based flash translation layer for solid-state drives," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*, Vancouver, BC, Canada, 2023.

[58] J. Sun, B. Reidys, D. Li, J. Chang, M. Snir, and J. Huang, "Fleetio: Managing multi-tenant cloud storage with multi-agent reinforcement learning," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'25)*, Rotterdam, Netherlands, 2025.

[59] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim, "Demystifying cxl memory with genuine cxl-ready systems and devices," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, Toronto, ON, Canada, 2023.

[60] 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)*. Oakland, CA: USENIX Association, Feb. 2018, pp. 49–66. [Online]. Available: https://www.usenix.org/conference/fast18/presentation/tavakkol

[61] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating System Support for Improving Data Locality on CC-NUMA Compute Servers," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VII, Cambridge, MA, 1996, pp. 279–289.

[62] S.-P. Yang, M. Kim, S. Nam, J. Park, J. yong Choi, E. H. Nam, E. Lee, S. Lee, and B. S. Kim, "Overcoming the memory wall with CXL-Enabled SSDs," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 601–617. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/yang-shao-peng

[63] M. Zhang, T. Ma, J. Hua, Z. Liu, K. Chen, N. Ding, F. Du, J. Jiang, T. Ma, and Y. Wu, "Partial failure resilient memory management system for (cxl-based) distributed shared memory," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, Koblenz, Germany, 2023.

[64] Y. Zhong, D. S. Berger, C. Waldspurger, R. Wee, I. Agarwal, R. Agarwal, F. Hady, K. Kumar, M. D. Hill, M. Chowdhury, and A. Cidon, "Managing memory tiers with CXL in virtualized environments," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 37–56. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong

APPENDIX

A. Abstract

We implemented SkyByte with a cycle-accurate simulator based on MacSim and SimpleSSD. This artifact includes the source code for SkyByte's simulation framework and detailed instructions for reproducing the key performance results presented in our paper.

This artifact can run on any x86 machine with at least 32 GB of RAM and 128 GB of disk space. For optimal performance, we recommend using a workstation with multiple high-performance CPU cores and at least 64 GB of RAM. The artifact requires a Linux environment (preferably Ubuntu 20.04 or later) and a compiler supporting the C++11 standard.

B. Artifact check-list (meta-information)

- **Algorithm:** The threshold-based context-switch trigger policy
- **Program:** Benchmarks from Rodinia, GAP, Splashv3, WHISPER, and DLRM. Their traces are included in the artifact.
- **Compilation:** g++ 11.4.0 or newer versions.
- **Model:** The meta DLRM model. Its traces are included.
- **Run-time environment:** Ubuntu 20.04 or newer versions.
- **Metrics:** Execution time, flash write traffic, and average memory access time.
- **Output:** Files and graphs, expected results included.
- **Experiments:** Generate experiments using provided scripts.
- **How much disk space required (approximately)?:** 128 GB.
- **How much time is needed to prepare workflow (approximately)?:** Around 40 minutes.
- **How much time is needed to complete experiments (approximately)?:** 3 days on a server with 32 CPU cores.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache-2.0.
- **Data licenses (if publicly available)?:** Apache-2.0.
- **Archived (provide DOI)?:** 10.5281/zenodo.14660184.

C. Description

1) How to access: The source code can be downloaded from Zenodo at https://zenodo.org/records/14660185. For the latest version, you can access our Github repo: git@github.com:platformxlab/skybyte.git.

2) Hardware dependencies: This artifact can run on any x86 machine with a minimum of 32 GB of RAM and 128 GB of disk space.

3) Software dependencies: The artifact requires a Linux environment (preferably Ubuntu 20.04 or later) and a compiler that supports the C++11 standard.

D. Installation

1) Start by downloading the SkyByte artifact from Zenodo:

```
1   wget https://zenodo.org/records/
        14660185/files/SkyByte-Artifact.tar.gz
2   tar -xvf SkyByte-Artifact.tar.gz
```

2) Please make sure all prerequisites are successfully installed:

```
1   sudo apt update
2   sudo apt-get install libboost-all-dev
3   sudo apt install scons htop
4   sudo apt upgrade g++
```

```
5   pip3 install matplotlib networkx pandas PyPDF2
        gdown scipy
```

3) Build the simulator for SkyByte:

```
1   cd SkyByte-Artifact
2   python3 build.py macsim.config -j NUM_THREADS
3   # e.g., python3 build.py macsim.config -j 30
```

E. Experiment workflow

This section describes the steps required to generate and execute the necessary experiments. We strongly recommend referring to *scripts-skybyte/README.md* for detailed explanations of each script used in this process.

Preparing the Multi-threaded Instruction Traces.

We provide the instruction traces captured using Intel's PIN tool for the workloads discussed in the paper. The traces can be downloaded from Zenodo:

```
1   wget https://zenodo.org/records/
        14660185/files/skybyte_new_traces.tar.gz
2   tar -xvf skybyte_new_traces.tar.gz
```

After extracting the files, ensure that the *skybyte_new_traces* folder and the codebase (the *SkyByte-Artifact* folder) are located in the same directory.

Each set of traces (e.g., the traces for bc with 16 threads) includes a trace configuration file (*trace.txt*) and several raw trace files (*trace_XX.raw*). The trace file format is consistent with that of the Macsim simulator. For more details, refer to Section 3.4 of *doc/macsim.pdf*.

Configuration Files.

The *configs* directory contains configuration files tailored for various workloads, design baselines, and specific settings (e.g., context-switch policies). For detailed information about these files, refer to *configs/README.md*.

Launching A Single Experiment.

After compiling the simulation framework, a symbolic link named *macsim* will appear in the *bin* directory. Within the same directory, the file *trace_file_list* specifies the location of the instruction trace configuration file (i.e., the corresponding *trace.txt*). This artifact includes scripts to automate the setup of individual experiments, which are described in a later section.

To launch a single experiment, use the following command:

```
1   cd bin
2   ./macsim -b ../configs/baselines/XX.config -w
        ../configs/workloads/XX.config (-t
        ../configs/settings/XX.config) -c {corenum} -o
        {terminal} -p -f {outputfile_name} (-d) (-r)
```

The command-line arguments are defined as follows:

```
1   -b baseline_setting_config_file_name
2   -w workload_config_filename
3   -t additional_setting_config_file_name (optional)
4   -c number_of_logical_cores_to_simlute
5   -o terminal_for_printing_warmup_logs (e.g. /dev/pts/6)
6   -p: print detailed runtime information (optional)
7   -f output_file_name
8   -d: run with infinite host DRAM (optional)
9   -r: output DRAM-only performance results (optional)
```

This command sets up the specified configurations (e.g., design baseline), performs a warmup, and replays the instruction traces on multiple simulated CPU cores and the simulated CXL-SSD. Results will be stored in the *output* directory.

Launching Batched Experiments.

To execute a large number of experiments simultaneously, we provide the *scripts-skybyte/run_all.sh* shell script. This script uses regular expressions to match multiple configuration files and automatically spawns experiments in separate `tmux` windows for parallel execution.

For convenience, we also provide the *artifact_run.sh* script, which automates the setup and execution of all required experiments. To launch all experiments, simply run:

```
1    ./artifact_run.sh
```

The variable `MAX_CORES_NUM` in the script specifies the maximum number of CPU cores allowed for simulations. Users may need to adjust this value based on their machine's specifications before running the script.

The *artifact_run.sh* script performs the following tasks: 1. Creates multiple directories named `bin-<workload>-<thread_num>-<baseline>` for different experiments. 2. Sets up the corresponding *trace_file_list* file in each directory. 3. Generates a *run_one.sh* script in each directory to facilitate running individual experiments. 4. Uses the *run_all.sh* script to launch parallel experiments.

See lines 23-29 of the *artifact_run.sh* script:

```
1    # Set up experiment configurations for figures 2, 3, 4,
         14, 15, 16, 17, 18, and Table 3
2    ./run_full.sh
3
4    # After running this, a folder named
         bin-<workload_name>-<thread_num>-<baseline_name>
         will be created for each experiment
5    # Inside each folder, there will be a script named
         run_one.sh to run the individual experiment
6
7    # Run experiments for figures 2, 3, 4, 14, 15, 16, 17,
         18, and Table 3 concurrently using multiple cores
8    ./run_all.sh -p "bc|tpcc|srad|radix|ycsb|dlrm|bfs-dense"
         -dr -j $MAX_CORES_NUM
```

F. Evaluation and expected results

To evaluate the artifact results, run the following command:

```
1    ./artifact_draw_figs.sh
```

This script collects all results from the *output* folder and generates the required figures sequentially. A detailed description of each command and the locations of the generated figures is provided within the script.

We provide the expected result data files and figures in the same directory where the figures will be generated. To verify the results, you can compare the generated figures directly with those presented in the paper, or compare the data for each figure with the example results we have provided.

G. Experiment customization

Custom Simulation Configurations.

In addition to the provided configurations, users can customize their own configuration files and evaluate them. Below is a list of configurable knobs that can be used to customize experiments:

1) `promotion_enable`: Enables or disables the adaptive page migration mechanism.
2) `write_log_enable`: Enables or disables CXL-Aware SSD DRAM management.
3) `device_triggered_ctx_swt`: Enables or disables the coordinated context switch mechanism.
4) `cs_threshold`: Defines the threshold for the context switch trigger policy. (Unit: ns)
5) `ssd_cache_size_byte`: Specifies the size of the SSD DRAM cache. (Unit: Byte)
6) `ssd_cache_way`: Defines the associativity of the SSD DRAM cache.
7) `host_dram_size_byte`: Specifies the size of the host main memory. (Unit: Byte)
8) `t_policy`: Defines the thread scheduling policy. Options include "RR", "RANDOM", and "FAIRNESS" (CFS).

Capturing Custom Program's Traces.

Users can generate custom traces for their own programs on their machines. To assist with this, we include a sub-repository called *macsim-x86trace* in the artifact. This sub-repo contains the Intel PIN 3.13 tool and scripts that generate both instruction traces and memory warmup traces, which are required by our simulation framework for a custom application. For detailed instructions on how to generate these traces, refer to *macsim-x86trace/README.md*. Please note that PIN 3.13 only runs on Ubuntu 18.04. Users may need to build a new OS environment for collecting custom traces.

H. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae