AstriFlash
A Flash-Based System for Online Services

Siddharth Gupta
EcoCloud, EPFL
siddharth.gupta@epfl.ch

Yunho Oh
Korea University
yunho_oh@korea.ac.kr

Lei Yan
EcoCloud, EPFL
l.yan@epfl.ch

Mark Sutherland
EcoCloud, EPFL
mark.sutherland@alumni.epfl.ch

Abhishek Bhattacharjee
Yale University
abhishek@cs.yale.edu

Babak Falsafi
EcoCloud, EPFL
babak.falsafi@epfl.ch

Peter Hsu
Peter Hsu & Associates
peter.hsu@phaa.eu

***Abstract*—Modern datacenters host datasets in DRAM to offer large-scale online services with tight tail-latency requirements. Unfortunately, as DRAM is expensive and increasingly difficult to scale, datacenter operators are forced to consider denser storage technologies. While modern flash-based storage exhibits μs-scale access latency, which is well within the tail-latency constraints of many online services, traditional demand paging abstraction used to manage memory and storage incurs high overheads and prohibits flash usage in online services. We introduce AstriFlash, a hardware-software co-design that tightly integrates flash and DRAM with ns-scale overheads. Our evaluation of server workloads with cycle-accurate full-system simulation shows that AstriFlash achieves 95% of a DRAM-only system's throughput while maintaining the required 99th-percentile tail latency and reducing the memory cost by 20x.**

I. Introduction

As billions of online users generate data daily, computer system designers struggle to architect datacenters that can manage ever-increasing datasets in a high-performance and cost-effective manner [11]. To provide online services with high throughput and low tail latency, modern datacenters host the majority of data in memory [21], [56], [58] using TBs of DRAM per server [10], [66]. Unfortunately, DRAM accounts for a significant fraction of the overall server cost [7] and is not scaling in capacity [11], [48], [52], [64]. Thus, datacenter operators are forced to consider denser technologies to host online services [22].

NAND flash is a suitable alternative as it enjoys 50x price ($/GB) improvement over DRAM [6], [34], but with 1000x higher latency [37], [44], [67], [80]. We believe that tighter integration of flash with DRAM might be a potential solution for two reasons. First, many modern online services have ms-scale end-to-end tail-latency constraints [18], [23], which allows them to absorb few μs-scale flash accesses [14], [18], [22], [41], [42], [44]. Second, object popularity and request distributions for datacenter workloads are inherently skewed [64], [73], [75], [76], thus allowing hosting the hot fraction of the dataset in DRAM that serves most requests and filters the bandwidth required from the backing flash. The above observations should permit the design of a cost-effective two-tier hierarchy where a capacity-constrained DRAM caches the hot fraction of the dataset stored in a capacity-scaled flash layer.

The central obstacle to such a design today is the reliance on the traditional OS abstraction of demand paging for moving data between memory and flash. While paging was originally introduced for devices with ms-scale access latencies (e.g., disks), modern flash-based devices provide $\sim$50 μs access latency. As a consequence, archaic OS paging mechanisms incur performance overheads unsuitable for the tight tail-latency constraints of online services [12], [19], [49], [50], [53]. Previous proposals combat the performance overheads by either accelerating paging [49], [50] or providing direct access to flash [1], [9], but still have a significant performance degradation compared to DRAM-only systems, or bypass paging and virtual memory altogether [41], [44], [65].

We propose *AstriFlash*, a hardware-software co-designed system that tightly integrates flash and DRAM to achieve DRAM-like performance with capacity and cost benefits of flash while maintaining the abstraction of virtual memory. We identify that paging overheads can be divided into core-side and memory-side arising from task switching and memory management. As a solution, we employ DRAM as a hardware-managed cache (e.g., Intel Knights Landing [68]) to eliminate the OS memory-management overheads and enable near-DRAM capacity management and data movement. We also hide the flash access latency using fast user-level thread switches triggered on a DRAM-cache miss instead of the traditional OS-based context switches, thus enabling efficient asynchronous flash accesses. While prior proposals [44], [49], [50], [65] typically focus on optimizing one class of overheads, AstriFlash achieves better performance by addressing core-side and memory-side overheads synergistically. Overall, AstriFlash efficiently absorbs the μs-scale flash latency by providing cross-stack integration with ns-scale overheads. Such integration requires a novel re-design of three essential techniques:

1) While switch-on-miss architectures have been studied for ns-scale memory stalls [17] and ms-scale disk stalls [71], AstriFlash requires absorbing μs-scale flash accesses in online services using flexible user-level threads. In contrast to a limited number of hardware threads and expensive OS-based

context switches, user-level threads enable low-cost contexts to efficiently overlap flash accesses by directly switching threads in 100 ns on a DRAM-cache miss. AstriFlash also provides hardware support to ensure forward progress and prevent starvation, thus upholding the tail latency of online services.

2) AstriFlash revisits classic proposals on memory traps [62] as it requires tolerating μs-scale DRAM-cache misses instead of rare ms-scale page faults using OS support. Accommodating frequent DRAM-cache misses in modern out-of-order (OoO) cores requires efficient microarchitectural support to revert committed stores residing in the Store Buffer (SB). AstriFlash extends existing speculation proposals [77] to cover the SB, thus allowing reverting stores without OS support.

3) AstriFlash avoids OS-based memory management using a hardware-managed DRAM cache and provides microarchitectural support for managing 100s of concurrent misses. While traditional on-chip cache designs implement costly SRAM-based structures to support 10s of concurrent misses for ns-scale cache-refill latency, the DRAM cache can have 100s of concurrent misses because of the μs-scale cache refills from flash. To the best of our knowledge, while previous DRAM-cache proposals [35], [36], [51], [63] do not provide such support, AstriFlash provides novel microarchitectural support to implement an in-DRAM miss status table to track concurrent misses at low cost.

Overall, AstriFlash provides a flash-based system for online services where a fast-but-expensive DRAM contains the hot fraction, and a slow-but-cheaper flash contains the dataset, thus reducing the memory cost by 20x. Our evaluation shows that AstriFlash achieves ~95% of the throughput of a DRAM-only system while maintaining the 99th-percentile latency.

II. Flash-Integrated Hierarchies

NAND flash offers 50x cost improvement [6], [34] but incurs 1000x higher latency (50 μs) than DRAM [37], [44], [67], [80]. While various online services with ms-scale tail latency constraints can absorb a few μs-scale flash accesses [14], [18], [22], [23], [41], [42], [44], replacing all DRAM with flash will result in unacceptable performance. Therefore, a careful combination of flash and DRAM is required to reduce costs while maintaining acceptable performance. Flash is commercialized as a storage device called Solid State Drive (SSD) with legacy I/O interfaces that preclude its use in online services. Therefore, tighter flash integration while maintaining DRAM-like performance requires the following considerations.

A. Identifying the required DRAM-to-flash ratio

Memory hierarchies are designed using caching principles to exploit the locality present in data accesses. Faster devices are used as a cache to serve frequently accessed data while backing slower devices serve data in case of a cache miss [47]. In datacenter workloads, a small fraction of the dataset can absorb most of the data accesses. Such an access pattern can be exploited by hosting the hot fraction of the dataset in DRAM while the backing flash contains the whole dataset [73], [74], [75], [76]. Each DRAM miss requires fetching the requested data from flash. Similar to previous studies [73], [76], we examine the DRAM miss ratio while varying the DRAM-to-flash capacity ratio for CloudSuite [23] workloads.

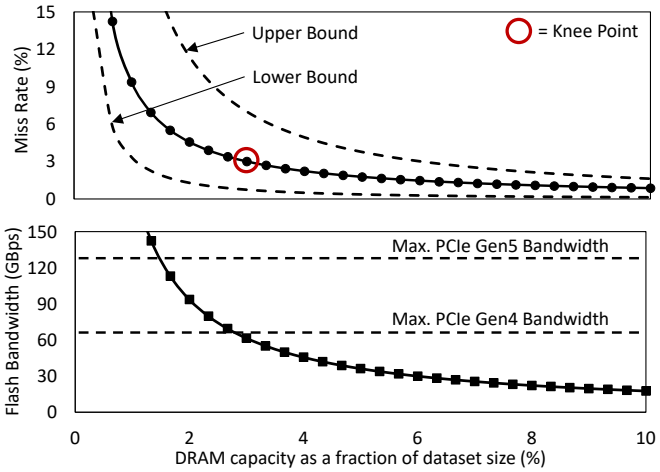


Fig. 1: Miss rate and flash bandwidth vs. DRAM capacity.

$$\text{BW}_{\text{Flash}} = \frac{\text{BW}_{\text{DRAM}}}{\text{Cache Block Size}} \times \text{Miss Rate} \times \text{Page Size} \quad (1)$$

We also study the tradeoff between DRAM capacity and flash bandwidth required to refill the DRAM. We calculate the flash bandwidth required per core using Equation 1 with 0.5 GBps as average DRAM bandwidth [74], [75], 4KB and 64B as the page and cache block size. Page is the smallest data unit in DRAM but is decided to be larger than the cache block to capture spatial locality in the lower levels of the memory hierarchy where temporal locality is scarce. Large pages also reduce the tracking metadata required for all pages in DRAM. Figure 1 shows the average cache miss ratio across workloads and the required flash bandwidth for different DRAM capacities. Similar to previous studies [73], [76], the miss rates flatten around 3% of DRAM capacity, which requires 60 GBps of flash bandwidth for a 64-core system [3]. With PCIe Gen5 specifications [20] providing up to 128 GBps bandwidth, it is possible to meet the flash bandwidth requirements for high core counts using multiple SSDs. To reduce the bandwidth requirements further, we can use a larger DRAM cache, employ smaller pages, or use optimizations such as Footprint Cache [36].

Henceforth, we assume a system with 1TB dataset hosted in flash and a DRAM cache with 3% capacity (i.e., 32GB), which requires 60 GBps aggregate flash bandwidth for 64 cores. As flash is 50x cheaper than DRAM, this configuration reduces the memory cost by 20x compared to a 1TB DRAM system [10], [66]. Today, we can already implement such a flash-based system with existing OS paging support. However, our workloads indicate that each core encounters a DRAM miss every 5-25 μs, thus causing the μs-scale paging support to become the performance bottleneck.

B. Programming abstractions for flash

Data movement between flash and DRAM layers can either be orchestrated by programmers or transparently done by the

OS using virtual memory [15]. While the latter places a lower burden on the programmer, its reliance on demand paging makes it more challenging to guarantee good performance. To stress-test AstriFlash, we do not expect the programmer to control data movement between the DRAM and flash layers.

Data movement in the case of virtual memory is transparent to programmers, and demand paging is used to retrieve data from flash. All pages are mapped into the application's Virtual Address (VA) space, but only a subset may be present in the Physical Address (PA) space, i.e., DRAM. When the application accesses a page not present in DRAM, the OS copies the required page from flash to DRAM and maps it to the correct VA. The application is oblivious to the physical location of a page and uses the same VA for each page throughout execution. Therefore, virtual memory provides ease of programmability by enabling permanent VA addresses for data.

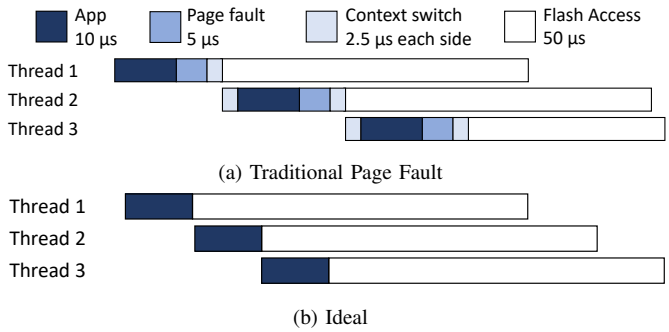
Data movement in case of explicit I/O is orchestrated by programmers. The application manages data transfer between flash and DRAM using system calls or user-space I/O [32], [44], [69], and can issue an explicit request to copy a page from flash into a self-managed memory pool. Complex indexing structures [44] are used to track pages in both DRAM and flash, and the limited capacity of the memory pool is exposed to the application. Thus, as DRAM pages are not guaranteed to remain at the same index throughout execution, the application cannot rely on permanent VA addresses.

C. Overheads of demand paging

Modern memory hierarchies use flash as a logical extension of memory. Demand paging abstractions use DRAM as an OS-managed cache for flash, where DRAM is exposed to the OS through a physical address space, and page tables map virtual addresses to physical addresses. When the application accesses a virtual address that is not mapped to a physical address, a page-fault exception is triggered, and the OS fetches the required page from flash and installs it in the physical address space, potentially evicting another page. After initiating the page fault, the OS also performs a context switch, thus overlapping the flash access with useful work [50].

Traditional demand paging abstraction for I/O was originally built for dealing with ms-scale access latencies (e.g., disks) where the device overheads overshadowed the μs-scale software overheads. However, as modern devices (e.g., flash) exhibit μs-scale access latencies, paging becomes a critical performance bottleneck [12]. We categorize these overheads into *memory-side* representing flash-access scheduling and memory-capacity management, and *core-side* representing task management and context switches.

Every page fault requires the OS to schedule an I/O request to fetch the requested page based on the SSD protocol (e.g., NVMe [57]). Checking the page cache and executing the OS storage stack and NVMe driver can consume up to 10 μs [9], [49], [50], [65]. Even with recent proposals that include lean software stacks [32], [69] or allow direct user access to flash using normal loads and stores [1], [9], systems still incur μs-scale overheads. As flash accesses are 1000x longer



(a) Traditional Page Fault

(b) Ideal

Fig. 2: Asynchronous flash accesses.

than DRAM accesses, frequently accessed pages should be maintained in DRAM for fast access. Migrating a page to DRAM might also require evicting another page, where victim selection is performed using complex policies in the OS with μs-scale overhead. Page migration also requires updating the address mappings in the page tables. Keeping the TLBs coherent with the page tables requires a global TLB shootdown, which removes the old entries from all the TLBs. Modern TLB shootdowns are a broadcast operation, thus scaling poorly with the number of cores and incurring over 10 μs in latency [4], [46]. Recent proposals [1], [46] attempt to reduce the overhead by batching multiple page faults together. However, for frequent flash accesses, the number of overall shootdowns grows with the core count, thus accumulating high memory-side overhead.

As OoO pipelines cannot hide 50 μs long flash access latency, the OS provides the abstraction of asynchronous flash accesses. The OS triggers a context switch on each page fault to overlap flash latency with useful work and maintain high system throughput. Previous proposals [1], [9] without context switches suffer a 5-10x performance degradation compared to DRAM-only systems. However, each context switch has $\sim$5 μs of core-side overhead [39], [65], [72] because of complex scheduling policies in the OS.

Figure 2 compares asynchronous flash accesses with traditional paging overheads and an ideal system with no paging overhead. With 3% DRAM capacity, modern datacenter workloads have a DRAM miss every $\sim$10 μs per thread accompanied by 10 μs of page fault and context switch overhead [65]. Moreover, TLB shootdowns and OS mechanisms result in global synchronization and thus do not scale with the number of cores. Clearly, μs-scale paging overhead causes high throughput degradation. To address these challenges, we propose AstriFlash, a hardware-software co-designed flash-based system for online services.

III. AstriFlash

This section describes the key insights for AstriFlash, provides an overview of its design, and discusses the tradeoffs against DRAM-only and flash-based memory hierarchies.

A. Key insights

AstriFlash 1) uses lightweight user-level thread switches to hide flash accesses and maintain tail-latency constraints,

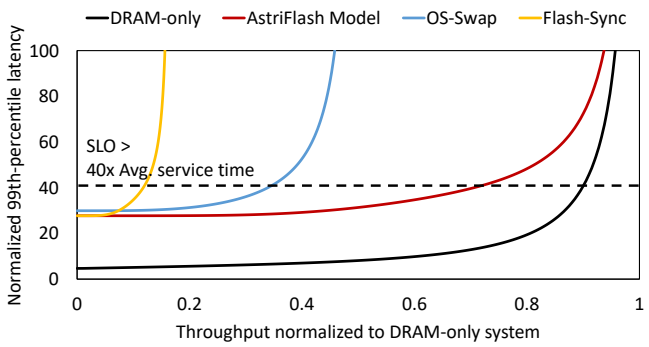


Fig. 3: 99th-percentile latency normalized to the average service time of a DRAM-only system with a single physical server.

and 2) eliminates traditional OS paging overheads using an accelerated DRAM miss handler.

The first insight aims at reducing the core-side overheads and is based on queuing characteristics at the tail of the response-latency distribution in online services with ms-scale Service-Level Objectives (SLO) [14], [18], [22], [23], [41], [42], [44]. The response latency of requests is dominated by queuing delay because a system that implements a single-server queuing model requires the younger requests to wait for older requests to finish and free up the server. With asynchronous flash accesses and lightweight thread-switching support, a single physical server can instead function as a logical multi-server queuing model. Therefore, older requests waiting for pending flash accesses can free up the server instead of blocking younger requests, thus removing the request-level head-of-the-line blocking and allowing AstriFlash to maintain similar overall response latency as a DRAM-only system. This insight applies best at high loads when there are multiple outstanding requests to cover the flash access latency.

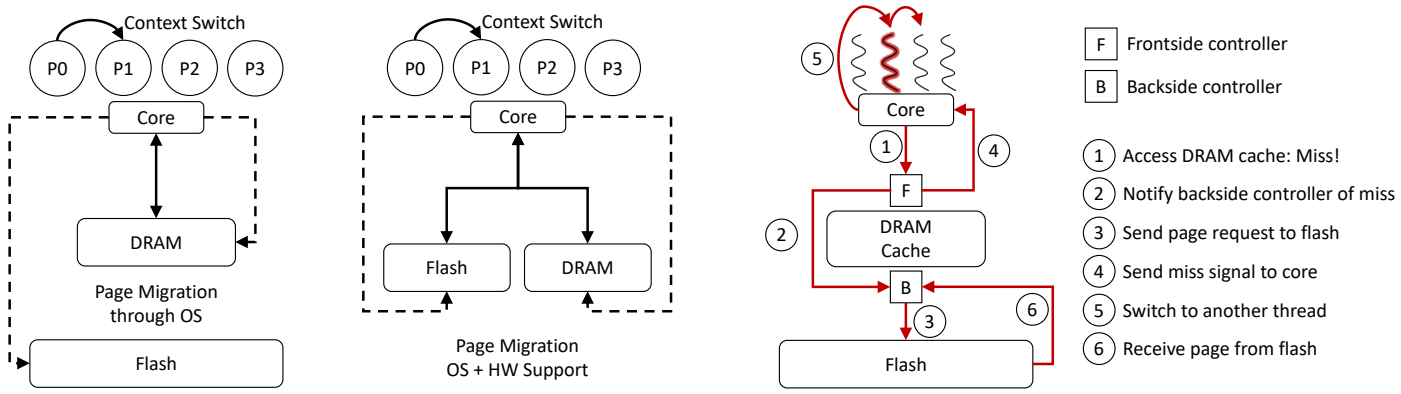
The second insight aims at eliminating the memory-side overheads and asserts that conventional OS demand paging abstractions are fundamentally not scalable in systems with high paging frequency because of synchronization in either software (i.e., multiple OS threads modifying kernel data structures like page tables) or hardware (i.e., broadcast-based TLB shootdowns), thereby limiting throughput as shown in Figure 2. Memory mapping flash and encapsulating the memory-management activity in an accelerated miss handler between the DRAM cache and flash enables removing the paging overheads and maximizing throughput.

Figure 3 presents the analytical latency and throughput of AstriFlash, a DRAM-only system, a traditional swap-based system (OS-Swap), and a system with synchronous flash accesses (Flash-Sync). DRAM-only and Flash-Sync represent an M/M/1 queuing system where all the requests always run to completion. In contrast, AstriFlash and OS-Swap represent an M/M/k queuing system where k requests are required to overlap the flash accesses. We assume that every 10 μs of execution triggers a flash access that takes 50 μs to complete. Flash-Sync has the worst performance with $>$80% throughput degradation as each flash access is synchronous. Though OS-Swap performs asynchronous flash accesses, it suffers from severe core-side and memory-side overheads (10 μs per flash access) because of page fault handling and context switches that cause $\sim$50% throughput degradation. AstriFlash has little overhead from flash access and thread switching, and thus the throughput approaches that of a DRAM-only system. Our analysis indicates that an application with flash accesses every $\sim$10 μs of execution requires a SLO of 40x the average service time to perform within $\sim$20% of the DRAM-only system. The performance gap shrinks as the SLO is relaxed or more physical servers are added. Overall, AstriFlash can sustain ms-scale SLOs for online services while achieving DRAM-like throughput using lightweight, user-level thread switching.

B. Design overview

Figure 4a, 4b, and 4c depict the design overview of OS-Swap, Flash-Sync, and AstriFlash respectively. On the core side, AstriFlash provides fast switching among user-space threads at a μs-scale granularity to efficiently overlap flash accesses with useful work. On the memory side, AstriFlash allows mapping flash into the physical address space while using DRAM as a hardware-managed cache (3% of the flash size).

1) Core-side design: AstriFlash provides novel hardware-software co-design [29] to hide asynchronous flash accesses efficiently. While OoO cores are provisioned to hide synchronous DRAM accesses, traditional disk accesses are long and infrequent enough to perform a context switch in the OS and hide the latency. However, in AstriFlash, we face μs-scale stalls which cannot be handled efficiently either in hardware or software [12]. Previous proposals [1], [76] expose the flash access latency to the cores and force them to wait for a response, thus losing throughput. In contrast, AstriFlash provides a novel *μs-scale switch-on-miss* architecture that allows switching user-level threads on a DRAM miss to hide flash accesses efficiently and achieve DRAM-like throughput. While switch-on-miss architectures [17] have been studied only in the context of ns-scale memory stalls and batch workloads like SPEC, AstriFlash needs to absorb μs-scale stalls and support latency-sensitive workloads. As online services can incur multiple outstanding flash accesses, flexible user-level threads are a cost-effective way to provide the traditional sequential execution abstraction to programmers and manage 100s of execution contexts [65], while hardware threads are limited in number. AstriFlash uses a simple user-level thread library [5], [61] to switch between threads in 100 ns, which is 50x faster than context switches, and 5x faster than recent proposals [39], [65]. AstriFlash also provides architectural support in OoO cores to invoke the user-level thread scheduler on a DRAM-cache miss, where the scheduler stops the running thread and schedules the next available thread. When the original thread is rescheduled after the flash access completes, it resumes from the instruction that caused the DRAM-cache miss and successfully reads the required data from the DRAM cache. AstriFlash also provides scheduling policies and architectural support to minimize request starvation.



(a) OS-Swap. (b) Flash-Sync. (c) AstriFlash.

Fig. 4: Comparison of hybrid memory system designs.

The μs-scale switch-on-miss design requires microarchitectural changes because the existing OoO cores assume that all successfully launched memory instructions finish synchronously. In contrast, on a DRAM-cache miss, AstriFlash aborts the responsible instruction, e.g., a committed store residing in the Store Buffer, and resets the core state to that of the last finished instruction before invoking the user-level thread scheduler. While a similar problem has been studied in the context of rare memory traps, previous proposals [62] rely on the OS to handle the trap. However, as DRAM-cache misses happen every $\sim$10 μs, AstriFlash cannot rely on the heavyweight OS mechanisms and requires microarchitectural support. As any memory instruction can potentially trigger a DRAM-cache miss and thus has to be aborted, instructions can only retire after the previous instruction completes, therefore disabling the performance critical memory reorderings from relaxed memory consistency models [55] and resulting in slow, sequentially-consistent execution. However, the memory reorderings can be performed speculatively if a rollback mechanism ensures correctness in case of aborts and discards all the speculative instructions. The speculation succeeds if the memory instruction completes, thus allowing the following speculatively-executed instructions to retire legally. Such speculation mechanisms [16], [24], [77] have been extensively studied in the context of memory consistency models and require extra tracking structures per core.

2) Memory-side design: As memory management and flash interaction contribute $\sim$5 μs of paging overhead per flash access, AstriFlash employs a hardware-managed DRAM cache to provide efficient near-DRAM capacity management and orchestration of data movement, thus eliminating the traditional OS paging overheads. Removing the explicit DRAM-capacity management in the OS also eliminates OS synchronization due to page-table modifications and TLB shootdowns for data movement between DRAM and flash, while substituting OS-based page replacement policies with cache eviction policies and I/O scheduling logic with hardware-triggered flash accesses. AstriFlash requires DRAM controllers to look up the DRAM cache and determine hit or miss decisions that are communicated to the requesting core, while the misses cause the corresponding pages to be fetched from flash.

Incorporating a GB-scale DRAM cache requires careful consideration in picking the page size. Traditional SRAM caches have 64B blocks as they need a small block size to track multiple independent data items to benefit from the temporal locality closer to the cores. In contrast, the DRAM cache being the last level in the cache hierarchy should be tailored for spatial locality as temporal locality is scarce. Moreover, having 64B blocks in a 32GB cache will require $\sim$4GB of tags which is impractical. Therefore, the DRAM cache in AstriFlash should employ a larger page size such as 4KB. However, even with a 4KB block size, we still need 64MB of tags which will be prohibitively expensive to hold in an SRAM tag array. Therefore, we conservatively employ designs that hold the tags in the DRAM cache at the cost of serialized tag and data lookup [35], [36]. System designers can pick alternative DRAM-cache designs [51], [63] if they can accommodate the required SRAM tag array.

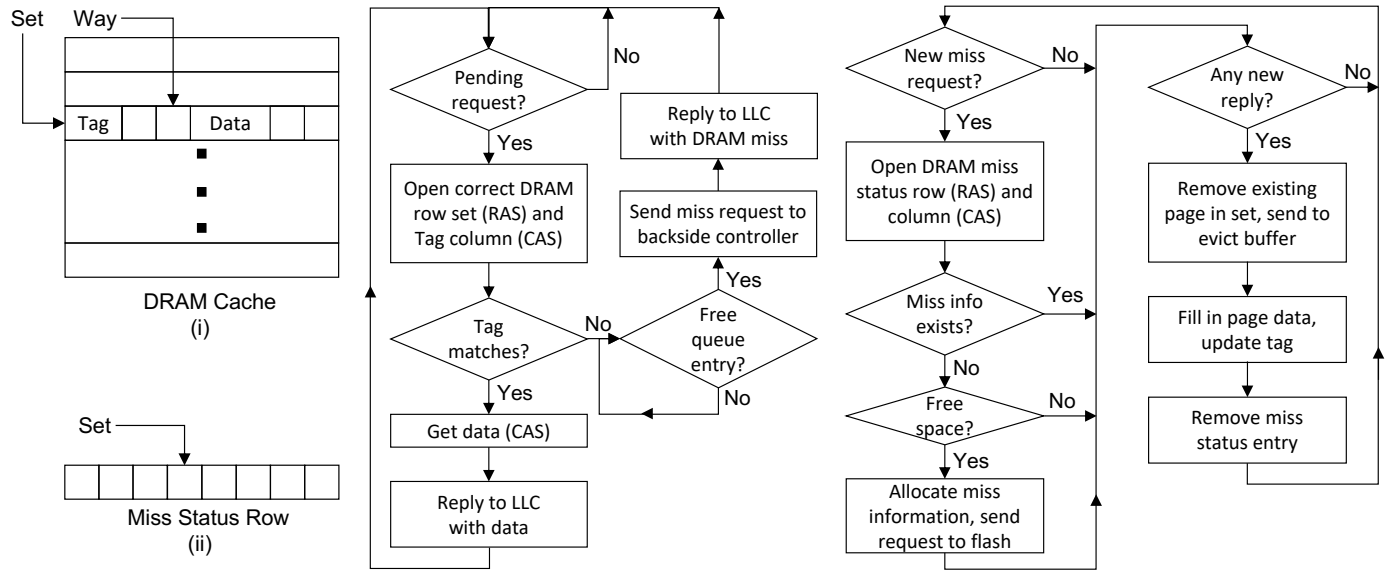
Similar to the non-blocking on-chip SRAM caches, the DRAM cache also needs to track multiple concurrent misses. While the on-chip caches use expensive SRAM structures (MSHRs) to track 10s of concurrent misses, the DRAM cache can have 100s of concurrent misses because of the long flash access latency and thus cannot rely on expensive SRAM structures. AstriFlash provides scalable bookkeeping of the DRAM-cache misses using hardware support for an in-DRAM miss status table, which is inexpensive compared to the typical CAM-based MSHR solutions. To the best of our knowledge, previous DRAM-cache proposals [35], [36], [51], [63] suffer from similar problems but do not provide any solutions.

IV. AstriFlash Implementation

This section describes the detailed implementation of AstriFlash mechanisms in the same sequence as required in the DRAM-cache miss-handling control path.

A. Flash addressing and memory mapping

AstriFlash uses existing PCIe mechanisms [1], [9] to create memory mappings for flash. PCIe devices have Base Address



(a) Physical organization. (b) Frontside controller. (c) Backside controller.

Fig. 5: AstriFlash DRAM cache structure and controller logic.

Registers (BARs) which contain the base address and offset of the address space assigned to an endpoint device such as an SSD. At boot time, the OS reads these BARs and acknowledges them as the physical address space assigned to the SSD. Then, the OS can use the page tables to directly map virtual addresses to these physical addresses, thereby exposing the SSD as a memory device. The obtained physical addresses are used to look up data in the cache hierarchy and are equivalent to the Logical Page Numbers of the SSD used internally in Flash Translation Layer and wear-leveling.

Address translation for servers provisioned with TBs of DRAM [10], [66] is an important problem [27] as modern TLB hierarchies cannot provide enough coverage. AstriFlash can benefit from previously proposed solutions [13], [27], [81], where Midgard [27] in particular is an excellent fit because it relies on large cache hierarchy capacity to reduce address translation overheads. AstriFlash and DRAM-only system have similar address translation overheads because both the hot data and corresponding page tables are served from the on-chip or DRAM caches. However, in the case of cold data accesses, both the data and the required page tables might have to be retrieved from flash. While AstriFlash's switch-on-miss architecture can provide DRAM-like throughput, serving page table from flash using a serialized page table walk might cause the application to violate its SLO. To address this challenge, AstriFlash employs available hardware and OS support to ensure that page tables are always DRAM resident.

On the hardware side, we use a hybrid-DRAM architecture similar to Intel's Knights Landing [68] in which the DRAM is split into a cache and a flat space that the OS can use directly. On the OS side, recent proposals [2] have engineered Linux to ensure page tables are allocated in specific DRAM nodes. Therefore, the OS can assuredly place page tables in a DRAM partition that is directly exposed to the OS. Such a design requires the DRAM controller to logically partition the available number of DRAM rows into flat and cached parts at boot time. The flat rows do not contain tags and expose a unique physical address range to the OS using BARs, while the cached rows contain tags and should be probed using the physical address range exposed by the flash BARs. For every memory access, if the requested physical address belongs to BARs dedicated to the flat rows, then the data is retrieved from the required flat row as per traditional DRAM design. Otherwise, the cached rows are searched for the required page using the set index and tag bits. Overall, AstriFlash can place page tables directly in DRAM and provide the address translation characteristics of a traditional DRAM-only system.

B. DRAM-cache organization

AstriFlash uses a DRAM-cache design [35] with 4KB page granularity instead of block-based caches [51], [63], therefore enabling a simple page fetch from flash on a DRAM-cache miss. Each DRAM row is equivalent to a set in a set-associative cache and contains both tag and data fields, as shown in Figure 5a. We implement two DRAM-cache controllers, where the *frontside controller* manages the DRAM-cache accesses, while the *backside controller* manages the DRAM-cache misses.

1) Frontside controller (FC): Figure 5b depicts that FC handles all DRAM data requests from the on-chip caches. It calculates hit/miss decisions by looking up the tags and sends replies for each request. We design FC by extending a traditional DRAM controller which inherits typical DRAM commands and scheduling policies. When FC receives a request, it calculates the set index (row number) from the address. It then checks if the row contains the page using a Row Address Strobe (RAS) operation to fetch the row into the DRAM row buffer,

followed by a Column Address Strobe (CAS) operation to fetch the tags and compare them against the requested address. The tags contain the physical addresses of the pages as exposed by flash and each tag occupies 8B. Therefore, each tag column (64B) can map up to 8 ways. If one of the tags matches, FC fetches the requested data with further CAS operations and sends it to the LLC. If no tag matches the requested address, FC sends a miss request to the backside controller's queue to fetch the requested page from flash. If the queue is full, FC stalls while waiting for free entries in the queue. Once the backside controller accepts the miss request, FC generates and sends a miss response for the data request to the LLC.

2) Backside controller (BC): Figure 5c depicts that BC interacts with flash and manages the metadata required for handling misses. As flash accesses are long (50 μs), BC has ample time to perform miss-handling operations. We propose implementing BC as programmable logic using microcode or software [25], [45] instead of hardwired FSMs, thus enabling the implementation of flexible and complex policies.

BC issues a 4KB read request to fetch a page from flash using the physical address. The on-chip network routes the requests generated by BC to the PCIe controller, which forwards them to flash [1], [9]. To receive the requested page, BC needs to secure available space in the corresponding DRAM set and might require evicting an existing page. After BC requests the page from flash, it identifies a victim page and copies it to the evict buffer. If the evicted page is dirty, it is written back to flash off the critical path. Once the requested page arrives, BC installs the data and tag in the designated set and way.

Traditionally, on-chip caches manage misses with Miss Status Handling Registers (MSHRs). As MSHRs are CAM-based and expensive, there are only tens of MSHRs per cache. In AstriFlash, as each miss lasts for 50 μs, it is possible to have hundreds of concurrent misses, which makes it too expensive to implement enough MSHRs. Instead, AstriFlash tracks the outstanding misses in a specialized DRAM row called *Miss Status Row (MSR)*. The MSR stores miss-handling entries containing the addresses and metadata of missing pages. To allow fast searches, we design the MSR as a set-associative structure where each entry is 8B and can be retrieved with a CAS operation. Once a DRAM-cache miss is detected, BC checks the MSR for a pending miss to the same page to avoid issuing duplicate requests to flash. If BC finds an existing entry, it discards the request; otherwise, it locates free entries in the set and allocates a new entry for the request. In case of no free entries, BC waits for pending flash requests to finish and free an MSR entry. Once the requested page arrives, BC removes the matching MSR entry, thus indicating miss completion.

In AstriFlash, the DRAM cache buffers all write operations that happen only on dirty page evictions, leading to fewer flash writebacks that are de-prioritized against reads. Flash writes are expensive as they can trigger garbage collection required for wear leveling, incurring up to 100ms of latency [26], [80]. To minimize garbage collection overheads, we suggest employing previous proposals [80] that perform block erasure only in the local plane, thus reducing wear-leveling latency.

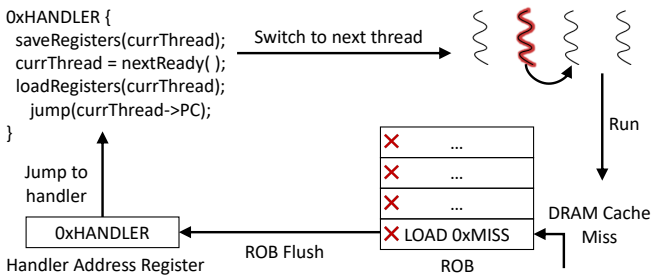


Fig. 6: Switch-on-miss hardware-software interface.

C. μs-scale switch-on-miss architecture

Once FC detects a DRAM-cache miss, it sends a miss signal to the core to trigger a thread switch instead of the traditional synchronous wait for data.

1) Sending a miss signal to the core: As the memory request corresponding to the DRAM-cache miss cannot be completed immediately and should be executed later, all resources allocated to the memory request should be reclaimed for the system to progress. E.g., if MSHRs retain memory requests that miss in the DRAM cache, all the MSHRs will eventually get occupied, and the on-chip caches will block. The mechanism required here is similar to the existing DRAM ECC error interface. When a non-correctable DRAM ECC error occurs, the DRAM controller generates an exception for the requesting core, and all the resources allocated to the request in the cache hierarchy are reclaimed [33], [70]. AstriFlash piggybacks miss signaling on the same mechanism, freeing up the allocated MSHRs at each cache level and sending the miss signal up the hierarchy towards the requesting core.

2) Triggering a thread switch: The miss signal triggers a user-level thread switch after reaching the core. Similar to previous proposals [30], [54], our mechanism for thread switching requires a new *Handler Address Register* and *Resume Register* as part of the architectural state. For each process, the handler address register contains the virtual address of the user-level handler, which will trigger a thread switch using the user-level thread library. For security purposes, this register can only be written in privileged mode, thus requiring an additional system call to verify and install a legitimate handler address. In contrast, the resume register can be read and written in user mode. Both these registers are part of the normal process state and will change on a context switch.

The core-side MSHRs track the memory requests sent to the cache hierarchy and can link the incoming miss signal back to the triggering instruction, which is assumed to be in the ROB as shown in Figure 6. Once the miss-triggering instruction is identified and all the older instructions have retired, the ROB is flushed and the Program Counter (PC) is set to the handler address. The PC of the miss-triggering instruction is saved in the resume register so that the thread can later resume from the same instruction. Thus, the CPU is not stalled for flash accesses and the control is passed back to the program.

3) Forward progress guarantees: The execution of multiple threads can cause a deadlock in AstriFlash. When rescheduling

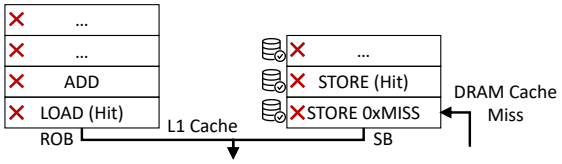


Fig. 7: Aborting speculative stores on a DRAM-cache miss.

a miss-generating thread, the requested page might have already been evicted because of DRAM-cache contention among multiple threads and cores, thus preventing the thread from progressing. To handle such cases, AstriFlash includes an architectural mechanism to force forward progress of threads when they are rescheduled so that they are not de-scheduled again before retiring at least one instruction. Therefore, even if the requested page is not present in DRAM when the thread is rescheduled, the thread is not switched out and blocks the core while waiting for the flash response. Moreover, AstriFlash can also expose such contention events to software for management at a higher level. AstriFlash implements this mechanism by adding a *forward progress* bit to the resume register. When a thread is rescheduled and the scheduler needs to force it to make forward progress, it stores the PC of the resuming instruction in the resume register and sets the forward progress bit. If this bit is set, the new memory request for the resuming instruction is forced to complete synchronously at FC, even if the DRAM cache misses. Therefore, the resuming instruction will block the core until it receives the requested data from the memory hierarchy, after which it retires and unsets the forward progress bit. Thus, the scheduler can flexibly choose between thread switches and forward progress while using contention information for thread scheduling or OS monitoring.

4) Precise exceptions and speculative stores: In modern processors, each core has a Store Buffer (SB) to collect stores that have retired but not completed. As stores do not produce direct register values for younger instructions, the store can be retired once its value and address are obtained and it is at the head of the ROB, and is sent to the SB where it awaits completion. In AstriFlash, as the DRAM cache might miss for store accesses, a thread switch will be triggered on corresponding stores. However, if the store has already retired from the pipeline and is resident in the SB, it cannot be discarded using existing speculation mechanisms. Therefore, as it is always possible for a store and the following instructions to be aborted due to a DRAM-cache miss, the OoO core is forced to run in a sequentially-consistent manner, thus prohibiting any memory reorderings of relaxed memory consistency models [55] and the non-speculative retirement of younger instructions. We employ post-retirement speculation techniques [16], [24], [77] that regain the performance of relaxed memory consistency models by speculatively reordering memory operations.

Based on ASO [77], we expand the speculation mechanisms already present in the ROB to cover the SB so that we can abort the "speculative" stores in case of a DRAM-cache miss. For typical ROB exceptions/speculation, each instruction's physical register mappings are kept until it retires from the ROB. In case of an exception or misspeculation, the core reverts to the older mappings before the instruction and discards newer mappings. We extend the speculation mechanism so that the mappings for a store are only freed when it leaves the SB. We assume a 4-way OoO ARM A76 core with 128-entry ROB, 32-entry SB, and a base 128-entry Physical Register File (PRF). We analyze our workloads to find that an average of four registers are modified between two stores, requiring four additional physical registers per store in the SB. Therefore, a 32-entry SB requires 32*4=128 additional registers in our PRF, equivalent to 1KB of additional SRAM. Finally, each store entry in the SB requires a map table to track the associated physical registers, where each map table entry represents 8-bit PRF indices for 32 registers, therefore requiring 32*32*8b = 1KB of SRAM. As PRF and map tables consume most of the additional silicon in ASO, we discount the silicon required for any additional microarchitectural structures. Based on 7nm SRAM density projections, modern silicon layouts can provide 2MB SRAM/mm^2 while Cortex A76 core is $1.3mm^2$ in size. Therefore, our 2KB overhead per core occupies $0.001mm^2$ (0.1% of Cortex A76), which might be acceptable.

D. Incorporating user-level threads

AstriFlash uses a user-level threading library that interacts with the hardware to provide hardware-triggered μs-scale thread switches on a DRAM-cache miss. The thread scheduler is designed to ensure that online services can satisfy their tail-latency requirements. We implement the proposed user-level threading library in C and Assembly and evaluate it with a cycle-accurate full-system simulator described in section V.

1) User-level threads: For each physical core, we assume a single global queue that receives jobs from the clients. The user-level scheduler picks new jobs from this queue and executes them on user-level worker threads. The same scheduler manages the context of each thread and switches among them for cooperative multithreading. This scheduling model [5], [19], [61] allows the applications to be easily ported to the AstriFlash infrastructure using the traditional notion of threads.

The scheduler cannot preempt the user-level threads directly as jobs are much smaller than the typical OS time quantum. Therefore, AstriFlash executes the jobs on a run-to-completion basis, except when they trigger a DRAM-cache miss and have to wait for data access from flash to complete. AstriFlash allows the user-level scheduler to be triggered on a DRAM-cache miss, which can be enabled by installing the scheduler handler's address in the handler address register. Once the scheduler is triggered, it deschedules the running thread that suffered the DRAM-cache miss and schedules a ready thread, thus overlapping the wait for the flash reply with useful work. The scheduler backs up the context for the running thread, consisting of the general-purpose registers and AstriFlash-specific resume register, and stores it on the thread stack. Logically, on a DRAM-cache miss, the running thread is halted and is stored in a pending job queue, as shown in Figure 8. The pending queue's size is limited so that pending jobs do not exceed the

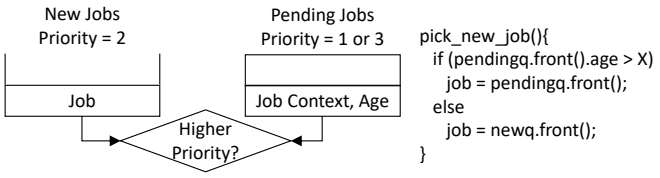


Fig. 8: Priority-based thread scheduling in AstriFlash.

tail-latency requirements. Once the pending queue is full, any new DRAM-cache misses result in the scheduler waiting for the flash response for the oldest job.

2) Priority scheduling with aging: The user-level thread library schedules new and pending jobs where the new jobs have never been scheduled for execution, while the pending jobs were halted after being scheduled because they suffered a DRAM-cache miss. The scheduler is optimized to provide the service latency distribution that matches the ideal Flash-Sync system (Figure 4b), where the jobs wait for flash to respond.

We implement a priority scheduler that assigns a default priority of one to pending jobs and a higher priority of two to new jobs to overlap the flash access latency with useful work. However, similar to traditional priority scheduling [71], AstriFlash also needs to handle the starvation problem. In a skewed job distribution, it is possible to have consecutive new jobs that do not face DRAM-cache misses, causing a simpler scheduler to starve the pending queue. However, Aging policies can be used to prevent such starvation. Each job entry records a timestamp when it enters the pending queue. When picking a new job to execute, the scheduler checks the head of the pending queue. If the age of the head job is greater than the average flash response time, then the scheduler picks it to execute; otherwise, it picks a new job, as shown in Figure 8. To combat latency spikes due to garbage collection and flash-side queuing, it is possible to program the backside controller and create a notification mechanism using queue pairs that can notify the core upon page arrivals from flash, similar to modern storage response arrivals [32], [69]. The scheduler can then read the queue pairs and schedule the corresponding thread. As datacenter jobs typically take 10-100 μs to finish, which is comparable to the flash latency itself, the pending jobs can be scheduled soon after their data arrives from flash. In this manner, the user-level thread scheduler maintains a service latency distribution similar to the Flash-Sync system.

V. Methodology

A. Applications and system architecture

We implement a user-level threading library that spawns 32-64 user threads per core (depending on the workload) and provides fast switching amongst them. We evaluate Silo and Masstree workloads from Tailbench [40] and port them to our threading library with few changes. We also evaluate five workloads from a microbenchmark suite [28] to capture data structure access patterns along with high-level database operations such as TATP and TPCC. In Array Swap, each operation swaps two array elements, generating both reads and writes. Red Black Tree (RBT) and Hash Table perform data structure lookups with pointer chasing behavior. TATP and TPCC execute 'update subscriber data' and 'neworder' transactions for items in a database. We model data accesses with an analytical Zipfian distribution so that the benchmarks trigger a DRAM-cache miss every 5-25 μs. Our workloads mimic limited write traffic as identified by previous proposals [1], [76], resulting in infrequent garbage collection events and practical endurance/lifetime for flash.

Core	16× ARM Cortex-A76 [79], 64-bit, 2GHz 4-way OoO, 128-entry ROB, 32-entry SB
TLB	L1(I,D): 48 entries, L2: 1024 entries
L1 Caches	64KB 4-way L1D, 64KB 4-way L1I 64-byte blocks, 2 ports, 32 MSHRs 2-cycle latency (tag+data)
LLC	1MB/tile, 16-way, 6-cycle access, non-inclusive
Coherence	Directory-based MESI
Interconnect	4 × 4 2D mesh, 16B links, 3 cycles/hop
DRAM cache	4 MCs, 2GB per MC (512MB per core) 128K sets, 4 ways, 16KB row, 4KB page RAS = 55 cycles, CAS (8B) = 3 cycles
SSD	50/100 μs random read/write latency 4KB page, 4GB/Plane, TLC, Plane-blocking GC

TABLE I: System parameters for simulation on QFlex.

We model 16× ARM Cortex-A76 cores with 1MB LLC per core. We scale down our 1TB dataset for 64 cores to a 256GB dataset for 16 cores and use an 8GB (3%) DRAM cache while flash stores the 256GB dataset. Both the DRAM cache and flash use the standard page size of 4KB. The frontside controller is an FSM that extends the traditional DRAM controller and uses FR-FCFS scheduling. We model one cycle each to issue commands for opening DRAM rows and columns, sending hit/miss responses to the on-chip caches, and miss requests to the backside controller. In contrast to the frontside controller, the backside controller is programmable and is slower in issuing requests. We model three cycles each for issuing DRAM commands and sending requests to flash.

For the AstriFlash scheduler, we implement job-based priority scheduling as described in subsection IV-D. We model a large job queue to evaluate the maximum throughput the system can sustain while also monitoring the service time of each job. Apart from actual work, the service time includes the wait time in case of a DRAM-cache miss but does not include the time spent waiting in the job queue. To measure the tail latency distribution, we use a Poisson process to model request arrival times and measure both the queuing time in the job queue and the service time.

B. Evaluated configurations

We use QFlex [59], a cycle-accurate full-system simulator based on Flexus [78] to evaluate AstriFlash. Table I lists the detailed simulation parameters used in QFlex. We evaluate the following configurations:

1) *DRAM-only* represents ideal performance, as all the data is served from DRAM without any flash accesses.

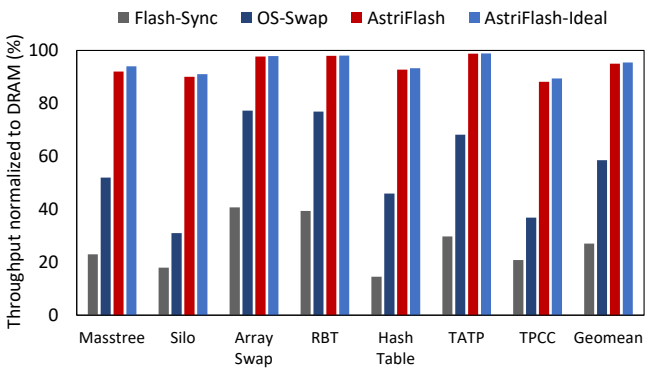


Fig. 9: Throughput comparison of different configurations normalized to a DRAM-only system.

2) *AstriFlash* represents our proposal, where DRAM acts as a hardware cache and contains the hot data, while the backing flash contains the whole dataset. A priority scheduler is used for switching among user-level threads, and each switch costs around 100ns.
3) *AstriFlash-Ideal* represents the AstriFlash design with no cost associated with thread switching.
4) *AstriFlash-noPS* represents AstriFlash with a FIFO scheduling policy instead of Priority Scheduling.
5) *AstriFlash-noDP* represents AstriFlash without DRAM partitioning, and thus TLB misses can incur flash-based page-table walks (subsection IV-A).
6) *OS-Swap* represents traditional systems where the OS uses paging to swap pages between the DRAM and flash (subsection III-A).
7) *Flash-Sync* represents FlatFlash [1], a latency-optimized system where the core waits for flash accesses to complete synchronously, thus resulting in a 50 μs delay.

VI. Evaluation

This section describes our evaluation of AstriFlash based on cycle-accurate simulation. The evaluation results include throughput, service time, and tail latency comparisons.

A. Throughput comparison

We first evaluate the throughput of AstriFlash compared to a DRAM-only system. Figure 9 shows that AstriFlash achieves an average of 95%, while AstriFlash-Ideal achieves 96% of the DRAM-only system's throughput. The 5% throughput loss is because of the DRAM-cache tag comparisons and response wait, pipeline flush for every DRAM-cache miss, and the scheduler overhead for switching threads.

For each DRAM-cache miss, DRAM-side tag checking is an overhead because it does not result in data access and will be repeated later. When the miss signal is sent to the core, the pipeline is flushed to redirect control to the user-level handler. As modern processors feature 100s of ROB entries, each flush loses useful work done by the OoO pipeline resulting in throughput degradation. As TPCC is the most computationally intensive workload, there is higher throughput degradation as each ROB flush is comparatively costlier. Finally, the user-level thread scheduler also causes throughput degradation as each switch on a DRAM-cache miss takes 100ns.

The OS-Swap configuration achieves 58% of the DRAM-only system's throughput as it has high page fault and context switch overheads for each DRAM-cache miss. The Flash-Sync configuration achieves only 27% of the DRAM-only system's throughput because the core has to wait for flash to respond to each DRAM-cache miss. Overall, AstriFlash achieves DRAM-like throughput while reducing the memory cost by 20x using a small DRAM cache and cost-effective flash.

Benchmark	AstriFlash	AstriFlash-noPS	AstriFlash-noDP
Masstree	1.02	4.61	2.43
Silo	1.01	6.73	1.57
Array Swap	1.04	12.03	1.84
RBT	1.03	5.98	2.03
Hash Table	1.01	3.92	1.26
TATP	1.01	15.33	2.01
TPCC	1.01	5.09	2.96
Geomean	1.02	6.82	1.76

TABLE II: Comparison of 99th-percentile service latency normalized to the Flash-Sync configuration service latency.

B. Service-latency comparison

AstriFlash must schedule both new and pending jobs fairly so that the service latency of the pending jobs does not lead to SLO violations. We show that AstriFlash – along with an optimized priority-based scheduling policy – achieves a similar latency distribution as in the Flash-Sync system. Table II compares the 99th-percentile latency from AstriFlash against AstriFlash-noPS that does not use the Priority Scheduler and AstriFlash-noDP that does not have DRAM partitioning, thus resulting in flash-based page-table walks. We normalize all latencies to the 99th-percentile latency of Flash-Sync as it represents the ideal latency when accessing flash. Compared to Flash-Sync, AstriFlash has a 2% latency degradation as once the requested page arrives, the non-preemptive scheduler might have to wait for the current job to finish before it can schedule the pending job. In contrast, AstriFlash-noPS has a $\sim$7x latency degradation as the scheduler executes new jobs even if the requested page for a pending job has arrived and only checks the pending queue when encountering a miss. Thus, the priority scheduler prevents the pending queue from starvation by checking it after every request. AstriFlash-noDP has a $\sim$70% latency degradation as page-table entries for cold pages have to be fetched from flash, which affects the 99th-percentile latency and can violate the SLO.

C. Tail-latency comparison

We study the tail-latency distribution for TATP, as it represents the short database operations present in datacenter workloads and takes ten μs on average. We use a Poisson process to model a bursty request arrival distribution and sweep the average inter-arrival time between requests from zero to ten μs, where each value represents a different request load in

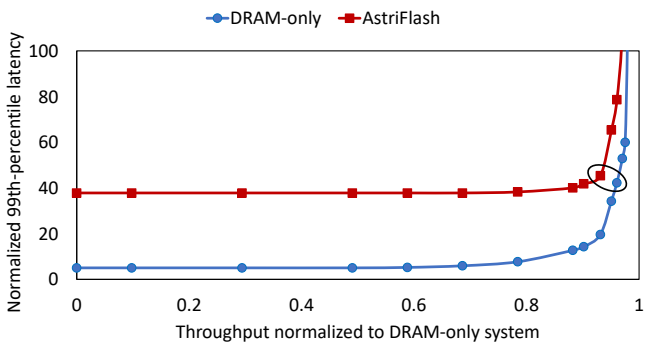


Fig. 10: Comparison of 99th-percentile latency normalized to average service time of a DRAM-only system.

the system. Figure 10 represents the tail-latency distribution of the DRAM-only system and AstriFlash. The X-axis represents the system throughput normalized to the maximum throughput of the DRAM-only system, while the Y-axis represents the 99th-percentile latency normalized to the average service time of the DRAM-only system, as previously shown in Figure 3. AstriFlash has higher 99th-percentile latency even at low loads with negligible queuing because of requests that require flash access. However, as the load increases, queuing latency also increases allowing the AstriFlash switch-on-miss architecture to overlap the flash access latency with queuing latency. Therefore, AstriFlash with 93% throughput matches the tail latency of a DRAM-only system with 96% throughput, thus maintaining the same tail latency with only 3% less throughput and 20x less memory cost.

The tail latency for the same load is worse because each DRAM-cache miss flushes the ROB, and the following thread switch destroys the on-chip cache/TLB locality, incurring more on-chip misses. It is also possible that requests which have already faced a long queuing delay further incur a DRAM-cache miss, thus exacerbating their overall response latency. Even though the service latency is higher as AstriFlash includes the flash access time, the queueing latency is significantly lower because fast DRAM-miss-triggered thread switches reduce the overall queueing, thus maintaining the same response time. Overall, as AstriFlash achieves near-DRAM 99th-percentile response latency, it enables online services to serve data directly from flash while maintaining the overall tail-latency constraints.

D. Garbage collection overheads

Garbage collection events in flash may block incoming read requests. For a flash with 256GB capacity, garbage collection blocks 4% of read/write requests [80], which impacts the tail latency. As AstriFlash uses a 1TB flash with more chips, the number of blocked requests reduces by more than 4x the 256GB flash, affecting less than 1% of the total requests. Moreover, AstriFlash can employ previously proposed local garbage collection algorithms to further enforce tail latency [80]. Finally, as flash writes are asynchronous, garbage collection generally happens off the critical path, thus preventing AstriFlash from suffering severe tail latency degradation.

VII. Related Work

AstriFlash is inspired by various previous proposals:

Flash integration: Flash-based memory systems significantly improve performance compared to disks. SSDAlloc [8] proposes a hybrid DRAM/flash memory manager while using flash as a log-structured page store. FlashMap [31] maps flash into a unified address space with DRAM. 2B-SSD [9] proposes accessing the same file with two separate byte-based and block-based I/O paths by utilizing a byte-addressable SSD and MMIO. FlatFlash [1] proposes a horizontally-tiered flash-based memory system to use DRAM as a cache for flash with customized page promotion techniques. A similar system was proposed by PageSeer [43] in the context of NVM. Overall, these proposals aim for a tighter integration of flash with the CPU.

Emerging memory technologies: NVM provides lower latency, higher bandwidth, and better endurance than flash [28]. Eisenman et al. [22] propose an NVM-based memory system for Facebook's workloads. They also study trade-offs across capacity, latency, and persistence and propose using NVM as a software-controlled cache between DRAM and flash. While NVM provides various attractive performance use cases, it does not benefit from economies of scale as of now and thus has severe volatility in its cost.

User-level threading and killer microseconds: AstriFlash's switch-on-miss architecture is inspired by informing memory operations [30] with lightweight multi-threading [54]. As lean user-level threading libraries [5], [44], [61] provide fast ns-scale thread switching, they are also useful for hiding DRAM accesses, e.g., co-routines for databases [38], [60]. Duplexity [53] shares resources in OoO cores with other smaller cores to hide μs-scale stalls, while Cho et al. [19] apply system-level modifications to get rid of μs-scale stalls. AIFM [65] uses highly-optimized user-level preemptions to migrate pages between local and remote devices efficiently.

VIII. Conclusion

We proposed AstriFlash, a flash-based system for online services that eliminates traditional demand paging overheads by employing a hardware/software co-designed μs-scale switch-on-miss architecture. AstriFlash serves data directly out of flash and achieves DRAM-like performance, thus allowing integration of denser and slower memory technologies for online services. Overall, AstriFlash reduces the memory cost by 20x, providing a solution for future TB-scale memory systems.

Acknowledgments

We thank Alexandros Daglis, Mario Drumond, Emilien Guandalino, Slim Fatnassi, and the anonymous reviewers for their feedback and support. This work was partially supported by FNS projects "Hardware/Software Co-Design for In-Memory Services" (200020B_188696) and "Memory-Centric Server Architecture for Datacenters" (200021_165749), a Qualcomm Innovation Fellowship (461760), and the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (2021R1G1A1094978).

REFERENCES

[1] A. H. M. O. Abulila, V. S. Mailthody, Z. Qureshi, J. Huang, N. S. Kim, J. Xiong, and W.-M. W. Hwu, "FlatFlash: Exploiting the Byte-Accessibility of SSDs within a Unified Memory-Storage Hierarchy." in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, 2019, pp. 971–985.

[2] R. Achermann, A. Panwar, A. Bhattacharjee, T. Roscoe, and J. Gandhi, "Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines." in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*, 2020, pp. 283–300.

[3] AMD EPYC, https://www.amd.com/en/products/cpu/amd-epyc-7742.

[4] N. Amit, "Optimizing the TLB Shootdown Algorithm with Page Access Tracking." in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, 2017, pp. 27–39.

[5] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism." in *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991, pp. 95–109.

[6] Anton Shilov, "Analysts Predict SSD Prices May Halve by Mid-2023," https://www.tomshardware.com/news/analysts-predict-ssd-prices-may-halve-by-mid-2023, 2022.

[7] AWS, "Amazon EC2 On-Demand Pricing," https://aws.amazon.com/ec2/pricing/on-demand, 2022.

[8] A. Badam and V. S. Pai, "SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy." in *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[9] D.-H. Bae, I. Jo, Y. Choi, J. Y. Hwang, S. Cho, D. D. G. Lee, and J. Jeong, "2B-SSD: The Case for Dual, Byte- and Block-Addressable Solid-State Drives." in *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*, 2018, pp. 425–438.

[10] J. Barr, "EC2 High Memory Update: New 18 TB and 24 TB Instances," https://aws.amazon.com/blogs/aws/ec2-high-memory-update-new-18-tb-and-24-tb-instances/, 2019.

[11] L. A. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[12] L. A. Barroso, M. Marty, D. A. Patterson, and P. Ranganathan, "Attack of the killer microseconds." *Commun. ACM*, vol. 60, no. 4, pp. 48–54, 2017.

[13] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers." in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 237–248.

[14] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A Protected Dataplane Operating System for High Throughput and Low Latency." in *Proceedings of the 11th Symposium on Operating System Design and Implementation (OSDI)*, 2014, pp. 49–65.

[15] A. Bhattacharjee and D. Lustig, *Architectural and Operating System Support for Virtual Memory*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2017.

[16] C. Blundell, M. M. K. Martin, and T. F. Wenisch, "InvisiFence: performance-transparent memory ordering in conventional multiprocessors." in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009, pp. 233–244.

[17] B. Boothe and A. G. Ranade, "Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors." in *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*, 1992, pp. 214–223.

[18] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services." in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, 2019, pp. 107–120.

[19] S. Cho, A. Suresh, T. Palit, M. Ferdman, and N. Honarmand, "Taming the Killer Microsecond." in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 627–640.

[20] A. Cunningham, "PCIe 5.0 is just beginning to come to new PCs, but version 6.0 is already here," https://arstechnica.com/gadgets/2022/01/pci-express-6-0-spec-is-finalized-doubling-bandwidth-for-ssds-gpus-and-more/, 2022.

[21] J. Dean and L. A. Barroso, "The tail at scale." *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[22] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. M. Hazelwood, C. Petersen, A. Cidon, and S. Katti, "Reducing DRAM footprint with NVM in facebook." in *Proceedings of the 2018 EuroSys Conference*, 2018, pp. 42:1–42:13.

[23] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware." in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVII)*, 2012, pp. 37–48.

[24] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is SC + ILP=RC?" in *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*, 1999, pp. 162–171.

[25] B. T. Gold, A. Ailamaki, L. Huston, and B. Falsafi, "Accelerating Database Operations Using a Network Processor." in *Proceedings of the 1st international workshop on Data management on new hardware*, 2005.

[26] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings." in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIV)*, 2009, pp. 229–240.

[27] S. Gupta, A. Bhattacharyya, Y. Oh, A. Bhattacharjee, B. Falsafi, and M. Payer, "Rebooting Virtual Memory with Midgard." in *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*, 2021, pp. 512–525.

[28] S. Gupta, A. Daglis, and B. Falsafi, "Distributed Logless Atomic Durability with Persistent Memory." in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 466–478.

[29] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Toward Dark Silicon in Servers." *IEEE Micro*, vol. 31, no. 4, pp. 6–15, 2011.

[30] 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 International Symposium on Computer Architecture (ISCA)*, 1996, pp. 260–270.

[31] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan, "Unified address translation for memory-mapped SSDs with FlashMap." in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 580–591.

[32] IO_URING, "Efficient IO with io_uring," https://kernel.dk/io_uring.pdf, 2020.

[33] B. L. Jacob, S. W. Ng, and D. T. Wang, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.

[34] James Carbone, "DRAM price increases will ease," https://electronics-sourcing.com/2022/05/12/dram-price-increases-will-ease/, 2022.

[35] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache." in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 25–37.

[36] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache." in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 404–415.

[37] Joel Hruska, "Why Latency Impacts SSD Performance More Than Bandwidth Does," https://www.extremetech.com/computing/325146-why-latency-impacts-ssd-performance-more-than-bandwidth-does, 2021.

[38] C. Jonathan, U. F. Minhas, J. Hunter, J. J. Levandoski, and G. V. Nishanov, "Exploiting Coroutines to Attack the "Killer Nanoseconds"." *Proc. VLDB Endow.*, vol. 11, no. 11, pp. 1702–1714, 2018.

[39] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency." in *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 345–360.

[40] H. Kasture and D. Sánchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications." in *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 3–12.

[41] A. Klimovic, H. Litz, and C. Kozyrakis, "ReFlex: Remote Flash $\approx$ Local Flash." in *Proceedings of the 22nd International Conference*

on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII), 2017, pp. 345–359.

[42] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic Ephemeral Storage for Serverless Analytics." in *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*, 2018, pp. 427–444.

[43] A. Kokolis, D. Skarlatos, and J. Torrellas, "PageSeer: Using Page Walks to Trigger Page Swaps in Hybrid Memory Systems." in *Proceedings of the 25th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2019, pp. 596–608.

[44] K. Kourtis, N. Ioannou, and I. Koltsidas, "Reaping the performance of fast NVM storage with uDepot." in *Proceedings of the 17th USENIX Conference on File and Storage Technologie (FAST)*, 2019, pp. 1–15.

[45] Y. O. Koçberber, B. Grot, J. Picorel, B. Falsafi, K. T. Lim, and P. Ranganathan, "Meet the walkers: accelerating index traversals for in-memory databases." in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 468–479.

[46] M. Kumar, S. Maass, S. Kashyap, J. Veselý, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "LATR: Lazy Translation Coherence." in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*, 2018, pp. 651–664.

[47] B. W. Lampson, "Hints for Computer System Design." in *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*, 1983, pp. 33–48.

[48] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative." in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009, pp. 2–13.

[49] G. Lee, W. Jin, W. Song, J. Gong, J. Bae, T. J. Ham, J. W. Lee, and J. Jeong, "A Case for Hardware-Based Demand Paging." in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 1103–1116.

[50] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong, "Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs." in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019, pp. 603–616.

[51] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked DRAM caches." in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 454–464.

[52] K. T. Malladi, F. A. Nothaft, K. Periyathambi, B. C. Lee, C. Kozyrakis, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile DRAM." in *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*, 2012, pp. 37–48.

[53] A. Mirhosseini, A. Sriraman, and T. F. Wenisch, "Enhancing Server Efficiency in the Face of Killer Microseconds." in *Proceedings of the 25th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2019, pp. 185–198.

[54] T. C. Mowry and S. R. Ramkissoon, "Software-Controlled Multithreading Using Informing Memory Operations." in *Proceedings of the 6th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2000, pp. 121–132.

[55] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence, Second Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[56] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA." in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 2014, pp. 3–18.

[57] NVM Express, *NVM Express Base Specification v1.4*. NVM Express Workgroup, 2019.

[58] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum, "Fast crash recovery in RAMCloud." in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011, pp. 29–41.

[59] Parallel Systems Architecture Lab (PARSA), EPFL, "QFlex," 2020. [Online]. Available: https://qflex.epfl.ch

[60] G. Psaropoulos, T. Legler, N. May, and A. Ailamaki, "Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins." *VLDB J.*, vol. 28, no. 4, pp. 451–471, 2019.

[61] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. K. Ousterhout, "Arachne: Core-Aware Thread Management." in *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*, 2018, pp. 145–160.

[62] X. Qiu and M. Dubois, "Tolerating Late Memory Traps in ILP Processors." in *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*, 1999, pp. 76–87.

[63] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design." in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 235–246.

[64] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology." in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009, pp. 24–33.

[65] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "AIFM: High-Performance, Application-Integrated Far Memory." in *Proceedings of the 14th Symposium on Operating System Design and Implementation (OSDI)*, 2020, pp. 315–332.

[66] S. Shah, "Announcing the general availability of 6 and 12 TB VMs for SAP HANA instances on Google Cloud Platform," https://cloud.google.com/blog/products/sap-google-cloud/announcing-the-general-availability-of-6-and-12tb-vms-for-sap-hana-instances-on-gcp, 2019.

[67] Skylake, "Intel Skylake," https://www.7-cpu.com/cpu/Skylake.html, 2019.

[68] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product." *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[69] SPDK, "Storage Performance Development Kit," https://spdk.io/, 2020.

[70] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly." in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)*, 2015, pp. 297–310.

[71] A. S. Tanenbaum, *Modern operating systems, 3rd Edition.* Pearson Prentice-Hall, 2009.

[72] D. Tsafrir, "The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)." in *Experimental Computer Science*, 2007, p. 4.

[73] D. Ustiugov, A. Daglis, J. Picorel, M. Sutherland, E. Bugnion, B. Falsafi, and D. N. Pnevmatikatos, "Design guidelines for high-performance SCM hierarchies." in *Proceedings of the International Symposium on Memory Systems (MemSys) 2018*, 2018, pp. 3–16.

[74] S. Volos, "Memory systems and interconnects for scale-out servers," *EPFL Thesis*, 2015. [Online]. Available: http://infoscience.epfl.ch/record/211040

[75] S. Volos, D. Jevdjic, B. Falsafi, and B. Grot, "Fat Caches for Scale-Out Servers." *IEEE Micro*, vol. 37, no. 2, pp. 90–103, 2017.

[76] F. A. Ware, J. Bueno, L. Gopalakrishnan, B. Haukness, C. Haywood, T. Juan, E. Linstadt, S. A. McKee, S. C. Woo, K. L. Wright, C. Hampel, and G. Bronner, "Architecting a hardware-managed hybrid DIMM optimized for cost/performance." in *Proceedings of the International Symposium on Memory Systems (MemSys) 2018*, 2018, pp. 327–340.

[77] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for store-wait-free multiprocessors." in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, 2007, pp. 266–277.

[78] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "SimFlex: Statistical Sampling of Computer System Simulation." *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006.

[79] Wikichip, "ARM Cortex A76," https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a76, 2020.

[80] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs." in *Proceedings of the 15th USENIX Conference on File and Storage Technologie (FAST)*, 2017, pp. 15–28.

[81] Z. Yan, D. Lustig, D. W. Nellans, and A. Bhattacharjee, "Translation ranger: operating system support for contiguity-aware TLBs." in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019, pp. 698–710.