AstriFlash: An Online Flash-Based Memory Hierarchy

Siddharth Gupta
EcoCloud, EPFL

Yunho Oh
EcoCloud, EPFL

Lei Yan
EcoCloud, EPFL

Mark Sutherland
EcoCloud, EPFL

Abhishek Bhattacharjee
Yale University

Babak Falsafi
EcoCloud, EPFL

Peter Hsu
Barcelona Supercomputing Center

ABSTRACT

Modern datacenters primarily host datasets in DRAM to offer large-scale online services with tight tail latency requirements. Unfortunately, DRAM is expensive and has slowed down in density scaling, forcing datacenter operators to consider denser storage technologies. While modern flash-based storage exhibits raw access latency in the μs scale which is well within the tail latency requirements of many online services, traditional OS abstractions and their implementation for paging from storage incur prohibitive overheads precluding flash's use in online services. In this paper, we introduce AstriFlash, a hardware/software co-design to integrate flash into the online memory hierarchy with a DRAM cache. Our evaluation with cycle-accurate full-system simulation shows that AstriFlash achieves 95% of DRAM-only system's throughput while maintaining 99-percentile tail latency with only 16% degradation, and reducing the overall memory cost by 20x.

1 INTRODUCTION

A critical challenge in modern datacenters is to manage the growing size of datasets given a budget constraint [9, 14]. To offer large-scale online services with high throughput and tight tail latency requirements, modern datacenters primarily host data in memory [5, 11, 29], requiring TBs of DRAM per server [20, 37]. Unfortunately, DRAM not only accounts for a substantial fraction of overall server cost, but it has also slowed down in capacity scaling [5] in recent years. DRAM cost and scaling challenges force datacenter operators to consider denser technologies to host online services.

NAND flash promises a two orders-of-magnitude cost per bit improvement compared to DRAM [16, 30]. Unfortunately, the current integration of flash into the memory hierarchy has prohibitive performance bottlenecks for online services. To begin with, raw flash access latencies are three orders of magnitude longer than DRAM, potentially allowing only services that can tolerate μs-scale latencies. More importantly, the Operating System (OS) abstractions for demand paging from storage devices not only exacerbate the incurred latency but also fundamentally limit the frequency of paging. We argue that such abstractions were built for *ms*-scale device access latencies in the 80's, where the software overhead was negligible. With modern device technologies and workload characteristics, storage access frequency is high enough that the OS abstractions quickly become a performance bottleneck [6].

We make several observations that together build a case for a careful integration of flash into a server node to host online services. First, many modern services have end-to-end tail latency constraints in the *ms* scale [8] and thus can absorb raw access latencies at the μs scale on the server side [7, 12]. Second, object popularity and request distribution for services are often highly skewed [41–43], allowing most data to be served directly out of a DRAM cache layer, thereby capping bandwidth demands on flash access. Third, online services can maintain response time for requests by overlapping asynchronous flash accesses to reduce queuing, which dominates the overall response time. Lastly, we can remove the traditional page fault overheads by accelerating the paging mechanism at the DRAM side in a centralized manner. Therefore, a platform can host services directly in flash if the hardware/software overheads do not dominate the flash access latency.

In this paper, we introduce AstriFlash, a hardware/software co-designed memory hierarchy, which can host online services directly in flash. AstriFlash maps the entire flash space into the user address space coupled with a hybrid DRAM cache to eliminate the legacy OS and CPU overheads for paging and address translation. AstriFlash uses DRAM as a transparent hardware-managed cache to serve CPU requests at high bandwidth while filtering accesses to flash. We propose an accelerated DRAM miss handler path, including microarchitectural support for DRAM miss notification, that eliminates legacy OS demand paging overheads. We design AstriFlash so that it achieves iso-DRAM throughput and maintains server-side tail latency constraints while reducing the overall memory cost by 20x. While we present AstriFlash with flash as the backend, the proposed architecture and mechanisms are readily applicable to other emerging memory technologies with μs-scale access latencies with varying cost and tail latency tradeoffs.

2 BACKGROUND AND MOTIVATION

In this section, we make the case that tighter integration of flash into the memory hierarchy would enable a significant reduction in cost while maintaining DRAM-like performance. We first introduce how to determine the sizes of DRAM and flash, based on analyzing data locality exhibited by server workloads. Then we explain traditional OS abstractions for utilizing flash devices and how they affect the performance in online services.

2.1 Tighter integration of flash

NAND flash is a technology that offers up to 50x cost improvement [16, 30] but incurs three orders-of-magnitude higher access latency (e.g., 50 μs) over DRAM [45]. While online services with μs-scale tail latency constraints may not be able to tolerate flash accesses [31], carefully architected services with *ms*-scale tail latency constraints

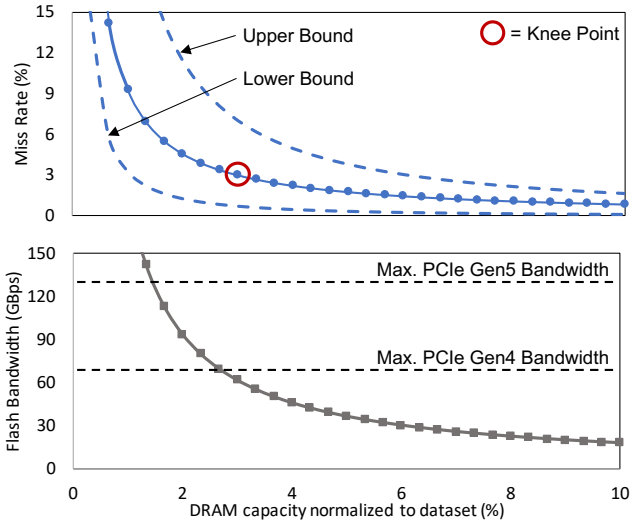


Figure 1: Miss rate and flash bandwidth vs DRAM cache size.

can absorb them [7, 23]. Traditionally, flash has been treated as a storage device, called Solid State Drive (SSD), with legacy I/O interfaces and OS abstractions.

2.1.1 Traditional storage systems. Memory hierarchies are designed using the principles of caching to exploit locality present in data accesses. Faster devices are used as a cache to host frequently accessed data while backing slower devices serve data in case of a cache miss [26]. As the on-chip cache subsystem uses synchronous cache accesses to get data, the OoO cores are designed accordingly to hide latencies of up to round-trip DRAM accesses.

Integrating flash devices in the memory hierarchy requires the abstraction of asynchronous accesses. As the core should not stall while waiting for long latency accesses, SSDs are accessed in the background to overlap data access with useful work. Traditional systems use SSDs as a logical extension of memory — e.g., through demand paging interface, while DRAM acts as a software-managed cache for SSDs. The OS provides the abstraction of asynchronous accesses where it allows the core to context switch to another process, therefore overlapping SSD access with useful work [27].

2.1.2 DRAM cache sizing. In datacenter workloads, a small fraction of the dataset exhibits a high degree of temporal data locality [41, 42]. Such an access pattern can be exploited by hosting the hot fraction of the dataset in DRAM, while the backing SSD contains the whole dataset [41, 42]. Each DRAM miss requires fetching the corresponding page from the SSD.

$$BW_{Flash} = BW_{DRAM} \times \frac{PageSize}{CacheBlockSize} \times MissRate \quad (1)$$

Based on previous studies [41, 43], we examine the miss ratio while varying the DRAM-to-flash capacity ratio for workloads in the CloudSuite benchmark suite [15]. To determine the required DRAM size, we studied the tradeoff between the DRAM capacity and the flash bandwidth required to refill the DRAM. We calculate the required flash bandwidth using Eq. 1 with 0.5 GBps as average DRAM bandwidth [42], 4KB as page size and 64B as cache block size. Figure 1 shows the representative average cache miss ratio along with the lower and upper bound, while also depicting the flash bandwidth required for various DRAM capacities. Similar to the previous studies [41, 43], the miss rates flatten around 2% to 4% of DRAM capacity, which requires 40-100 GBps of flash bandwidth for a 64 core setup [2, 17]. As specifications for PCIe Gen5 [34] report up to 128 GBps of bandwidth, we can easily meet these bandwidth requirements for high core counts. In case of bandwidth constrained servers, we can use a bigger DRAM cache which correspondingly reduces the flash bandwidth requirements.

For our study, we consider the DRAM capacity of 3%, which requires 0.9 GBps of flash bandwidth per core, and 57.6 GBps aggregate bandwidth for 64 cores. Overall, we assume a system with 1 TB dataset per server hosted in SSD with a 3% DRAM cache capacity (i.e., 32 GB). As SSD is up to 50x cheaper than DRAM [16, 30], such an arrangement can reduce the memory cost by 20x compared to 1 TB DRAM system [37]. We can already implement such a flash-based memory hierarchy with existing OS abstractions and hardware mechanisms. However, based on the calculated miss rates and bandwidth requirements, as each core has a DRAM miss every 5-25 μs, the OS abstractions themselves become a critical performance bottleneck.

2.2 Demand Paging vs Explicit IO

There are two main techniques with which systems utilize the storage devices (SSD in our case), demand paging and explicit IO. In demand paging, the application does not directly interact with the SSD, but instead uses the abstraction of virtual memory. All pages in use by the application are mapped into its Virtual Address (VA) space, while only a subset of them might be present in the Physical Address (PA) space i.e. DRAM. When the application accesses a page that is not present in DRAM, the OS fetches the required page from SSD and maps it to the correct VA. Instead, in explicit IO, the application manages the transfer of data between SSD and DRAM using system calls or user-space IO [18, 24, 39]. Thus, whenever the application requires a page from SSD, it issues an explicit request to read the corresponding page from the SSD into a memory pool managed by the application itself.

The key difference between these two techniques is the use of virtual memory. In demand paging, the application is oblivious to the actual physical device each page resides in. It also uses the same VA for each page throughout the execution which provides ease of programmability, for e.g. it allows pointers to other pages in the same VA space. In contrast, explicit IO requires the application to track pages present in both DRAM and SSD at any point in time, which is done through complex indexing structures [24]. The DRAM capacity is exposed to the application as the memory pool is of a limited size. Thus, the pages present in DRAM cannot generally maintain the same VA throughout the execution, which also prevents them from using direct VA pointers. Therefore, in this paper, we focus only on the demand paging technique. However, it is possible to extend AstriFlash to provide explicit IO as well. The DRAM can be divided into two parts: one of which is controlled by the hardware as a cache, while the other can be used as a normal DRAM space controlled by the software [38], which can thus be used for explicit IO.

2.3 Bottlenecks in traditional OS abstractions

The flash-based memory hierarchy relies on OS support to use SSD as a logical extension of DRAM. With the virtual memory abstraction, the OS uses DRAM through a physical address space and uses page tables to map the process-specific virtual pages to DRAM, while marking the pages present in the SSD as swapped-out. On accessing a virtual address, if the requested address is not mapped to DRAM, the control goes to the OS which then fetches the required page from SSD using the abstraction of a page fault.

2.3.1 IO access scheduling. Each page fault requires accessing a page from SSD, for which the OS schedules an IO request based on the SSD protocol, such as NVMe [13]. Checking the page cache, traversing the overall OS storage stack, and NVMe driver accumulates overhead of up to 10 μs. Recent software solutions include lean software stacks [18, 39], while hardware proposals focus on reducing this overhead by memory mapping the SSD in the user address space, thus allowing the user to directly access SSDs using normal loads and stores [1, 4].

2.3.2 Page migration and eviction. Because SSD accesses have 1000x longer latency than DRAM access, frequently accessed pages should be migrated and kept in DRAM for fast access. Migrating a page to DRAM also requires evicting another page to make space, which is done through complex page selection policies in the OS with μs-scale overhead. Apart from that, each page migration requires modifying the virtual-to-physical address mappings in the page tables. Keeping the TLBs coherent with the page tables requires a global TLB shootdown, which removes the modified entries from all the TLBs. As this TLB shootdown operation is a broadcast, it does not scale with the number of cores and incurs high overhead. Previous studies report that TLB shootdown latencies can be over 10 μs [25] and try to reduce the overall TLB shootdown frequency by batching multiple page faults together [1, 25]. However, in case of frequent SSD accesses, the number of overall shootdowns grows with the core count, therefore accumulating high overhead.

2.3.3 Asynchronous SSD acceses. OoO pipelines cannot hide SSD accesses as each access takes around 50 μs. Therefore, to overlap the wait for SSD response with useful work, the OS provides the abstraction of asynchronous SSD accesses along with context switches. On a long latency IO access, the OS performs a context switch, scheduling another independent process/thread which can perform useful work, thus maintaining high system throughput. Previous studies have quantified the overhead of context switches to be around 5 μs [10, 40]. As previous flash-based memory systems [1, 4] do not focus on maintaining the asynchronous access abstractions, they have 5-10x performance difference from DRAM-only systems.

The traditional OS abstractions were initially built for dealing with *ms*-scale IO access (e.g., disks). As the IO devices were much slower than the CPU and accesses were infrequent, the μs-scale software overheads were negligible. But as modern storage devices, such as SSDs, exhibit μs-scale access latencies, the software overhead from traditional abstractions becomes a critical performance bottleneck [6].

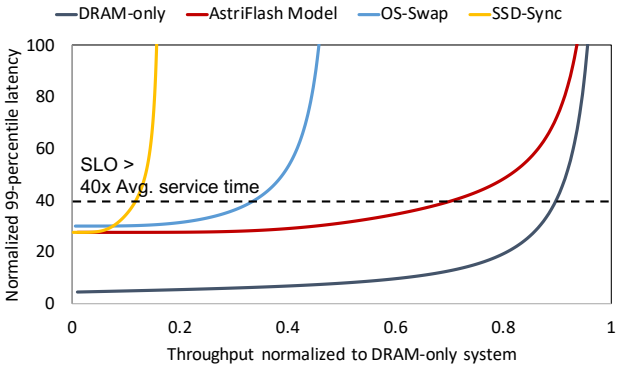


Figure 2: 99-percentile latency normalized to average service time of DRAM-only system in a single server.

3 ASTRIFLASH

In this section, we first describe the key insights underlying the AstriFlash architecture and the throughput vs. tail latency tradeoffs in DRAM-only and AstriFlash systems. We then present a high-level overview of the key mechanisms we propose to achieve the desired throughput and tail latency.

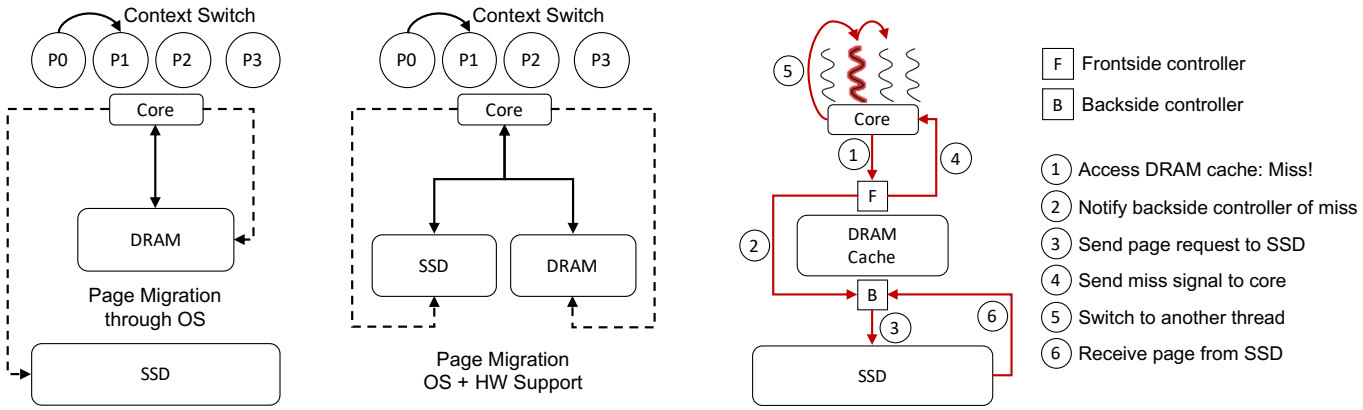
3.1 Key insights

AstriFlash is designed based on following two key insights: 1) overlapping multiple SSD accesses with lightweight thread switches enables maintaining tail latency constraints, and 2) eliminating overheads in conventional OS abstractions for paging and accelerating a DRAM miss handler path helps maximize throughput.

The first insight is based on the queuing characteristics at the tail of the response distribution in online services with *ms*-scale tail latency [7, 22]. The overall response time of these requests is dominated by queuing time. Assuming a single server queuing model (in a DRAM-only system), these requests wait in queue for older requests to finish and free up resources. In a system with asynchronous SSD accesses and lightweight thread switching, one should be able to architect the online services to behave like a multi-server queuing model, thereby replacing the wait for older requests with the wait for an SSD response and enabling the online service to absorb SSD accesses while maintaining the same overall response latency.

The second insight is that conventional OS abstractions for paging not only incur high overhead, they are fundamentally not scalable in systems with high paging frequency (e.g., multi-core CPUs with multiple threads/core), thereby limiting throughput. These overheads both slow down paging and limit scalability because of synchronization in either software (i.e., accesses from multiple OS threads to shared data structures and page tables) or in hardware (i.e., broadcast-based TLB shootdown). Mapping the SSD into user address space and encapsulating the paging activity in an accelerated miss handler path between a DRAM cache and SSD enables removing page fault overheads and maximizing throughput.

Fig. 2 captures the analytical throughput and latency characteristics of AstriFlash, along with DRAM-only system, traditional OS



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

Figure 3: Comparison of hybrid memory system structures.

swap-based hierarchy (OS-Swap) and synchronous SSD accesses (SSD-Sync). On one hand, the ideal DRAM-only and SSD-Sync systems represent an M/M/1 queuing model as all the requests execute in a run-to-completion mode. On the other hand, OS-Swap and AstriFlash can be modeled as an M/M/k queuing system where k requests are required to overlap the SSD accesses. SSD-Sync has the worst performance, as each SSD access is synchronous and accompanied by 50 μs of wait time. Though OS-Swap performs asynchronous SSD accesses, it suffers from page fault handling/context-switch overheads (10 μs), therefore causing 50% throughput degradation. As AstriFlash claims very little overhead per DRAM miss to switch to another job, the overall throughput and tail latency approach that of a DRAM-only system. However, our analysis indicates that only services with SLOs of greater than 40x service time will be able to absorb SSD accesses and thus benefit from AstriFlash. Overall, AstriFlash does not violate SLO for classes of online services with *ms*-scale tail latency requirements while achieving iso-DRAM throughput through lightweight user-level switching.

3.2 Design overview

Fig. 3a, Fig. 3b, and Fig. 3c depict the OS-Swap, SSD-Sync, and AstriFlash configurations respectively. AstriFlash maps the entire flash space into the user address space and uses DRAM as a transparent hardware-managed cache. The AstriFlash memory hierarchy accesses SSDs like a memory device instead of traditional OS-level paging, thus eliminating the OS-side overhead for page fault handling, I/O scheduling, page table modifications, and TLB shootdowns. AstriFlash employs a page-based DRAM cache [21] as it outperforms block-based caches [28, 33] in case of server workloads because of spatial locality advantages. Moreover, a page-based cache matches the SSD interface because the minimum page size of DRAM and SSD are the same (4KB in x86-64 architecture). Based on the analysis in Sec. 2.1, DRAM cache is 3% of the SSD size.

The key design contribution for AstriFlash is a fast 'switch-on-miss' architecture, which switches user-level threads on a DRAM cache miss to efficiently hide the SSD access latency and achieve iso-DRAM throughput. Previously proposed horizontally-tiered memory systems [1, 43] expose the SSD access latency to the cores, forcing them to wait for SSD response, thus losing throughput.

Modern OoO cores are provisioned to hide synchronous DRAM accesses in case on-chip caches don't have the required data. On the other extreme, accesses to storage devices are traditionally slow enough to let the software take control and set up data access along with a context switch. However, in the case of AstriFlash, we face μs-scale stalls, which cannot be handled efficiently either in hardware or software [6]. The proposed switch-on-miss architecture is a fundamental change in instruction execution semantics because the traditional cores and memory hierarchy are designed for synchronous accesses, which means all loads successfully launched by the cores are guaranteed to finish.

The proposed switch-on-miss architecture incorporates hardware-supported asynchronous SSD accesses and fast hardware-triggered user-level thread switches. As typical datacenter workloads will incur more DRAM misses while a page is accessed from SSD, user-level threads provide a cost-effective way to manage multiple execution contexts along with quick switching to efficiently overlap the SSD accesses. Thus, user-level threads far outweigh using hardware threads which are limited in number and do not provide the required number of contexts for our scenario. Therefore, an application that does not utilize user-level threads will be limited by the parallelism provided by the hardware threads, and will typically have to wait for the SSD accesses to complete synchronously.

To support such mechanisms, AstriFlash includes a new pair of frontside/backside DRAM cache controllers, as shown in Fig. 3c. The frontside controller is responsible for handling the core-side interactions, detecting a DRAM cache hit or miss, and sending a DRAM cache miss signal to cores. The backside controller is responsible for managing the DRAM cache miss handling information and handling SSD page access requests/responses.

3.3 Switch-on-miss architecture

AstriFlash requires the workloads to use a user-level thread library [3, 32] with which they can switch between threads with low overhead. The user-level threads also abstract away the asynchronous interface provided by AstriFlash hardware and provide the traditional

abstraction of sequential execution to programmers. On a DRAM cache miss, while fetching the data from SSD, AstriFlash halts the current memory instruction and requests a thread switch from the user-level thread library. When the original thread is rescheduled, it resumes from the memory instruction which caused the DRAM miss. Because of the SSD access which completed asynchronously, the thread can access the required data directly from the DRAM cache. This model allows the applications to be easily ported to the AstriFlash infrastructure using the traditional notion of threads. To implement the described switch-on-miss architecture, we propose a hardware-software co-designed user-level handler which is triggered by the hardware on each DRAM miss, and performs backup and restore of thread contexts, thus allowing switching between threads. The workload needs to provide the address of this user-supplied handler in additional specialized registers for the hardware to trigger the handler on DRAM misses.

We also propose an AstriFlash-specific priority scheduling mechanism to maintain the service latency of each request. We design the scheduler to optimize for the service latency distribution of the jobs so that it is similar to the ideal distribution of the SSD-Sync (Fig. 3b), where the core waits for the SSD to respond. On a DRAM miss, AstriFlash prioritizes new jobs over pending jobs so that it can overlap the latency of fetching the page from SSD with useful work from the new jobs. However, in a skewed distribution, it is possible to have a consecutive series of jobs that do not face any DRAM miss, causing the priority scheduler to starve the pending queue. We use aging techniques to solve such starvation problems, in which case the jobs that have been waiting for longer than the SSD response latency are promoted to a higher priority than the new jobs. This allows the AstriFlash scheduler to maintain service latencies similar to the SSD-Sync configuration. Overall, the proposed switch-on-miss architecture enables thread switching within 100 ns, which is 10x faster than current OS-level switches.

3.4 Implementation Highlights

In this subsection, we briefly describe some of the implementation details of AstriFlash.

3.4.1 Address Space. In AstriFlash, the SSD is exposed as a memory device to the CPU using the PCIe base address registers [1] which can be setup during boot time. Thus, the OS can see the SSD as a physical address space, and thus can map the required virtual address pages. In this manner, the SSD acts as the lowest level in the memory hierarchy.

3.4.2 Partitioned DRAM Cache. In AstriFlash, as the DRAM is used as a hardware-managed cache, the software does not control the contents in the cache. In some cases, critical OS metadata might be evicted from the DRAM cache, and thus has to fetched from the SSD, slowing down the OS. For e.g., the page tables containing the mapping for a cold page might also reside in SSD, thus requiring a page table walk out of SSD (multiple pages) before the page itself can be retrieved. To prevent such situations, we propose to have a small DRAM partition (2-4 GB) separate from the DRAM cache [38], which exposes its own physical address space and thus can be directly managed by the OS. This DRAM partition can be used to keep critical OS data-structures like page-tables, thus always guaranteeing fast access.

3.4.3 Miss Handling. Traditionally, on-chip caches manage misses with CAM-based structures called Miss Status Handling Registers (MSHRs), which are used to track outstanding misses, and prevent duplicate requests for the same blocks. As these MSHRs have substantial hardware overhead, the on-chip caches have only 10s of MSHRs. However, in AstriFlash, as each miss lasts for 50 μs, there can be 100s of misses happening in parallel, which makes it too expensive to implement enough MSHRs. Instead, AstriFlash manages the miss handling information in a specialized DRAM row called Miss Status Row (MSR). The MSR is a set-associative structure which can easily track 100s of outstanding misses, and poses negligible latency compared to the SSD response time.

3.4.4 DRAM Cache Miss Notifications. Once a miss has been detected on the DRAM cache side, the core has to be notified of the miss so that it can switch threads while the requested page is fetched from the SSD. This also requires deallocating all the hardware resources allocated to the corresponding memory access in order to prevent stalls. E.g., if all MSHRs are full of memory requests that miss in the DRAM cache, all the on-chip caches will eventually block. Such a mechanism is similar to the existing DRAM ECC error interface. When a non-correctable error occurs, the DRAM controller generates an exception to inform the software of the error. Such errors also require removing all traces of the request from the cache hierarchy [19, 36]. AstriFlash piggybacks the miss signaling on the same mechanism, which frees up the allocated MSHRs at each cache level and sends the miss signal up the hierarchy towards the requesting core.

3.4.5 Forward progress guarantees. While AstriFlash supports an efficient switch-on-miss mechanism, executions of multiple threads may cause a livelock problem. When rescheduling a switched-out thread, the requested page might have already been evicted because of DRAM cache contention between multiple threads, preventing the current thread from progressing. To handle such cases, AstriFlash includes a microarchitectural mechanism to guarantee forward progress of threads once they are rescheduled. When a thread is rescheduled, the user-level scheduler restores the PC to the instruction on which the DRAM miss was detected. AstriFlash maintains a forward progress bit indicating if the previous DRAM miss instruction has been successfully committed or not. If this bit is set, a new miss signal cannot cause a thread switch on the same instruction where the DRAM miss was detected previously. Therefore, this instruction blocks the core until it receives the requested data from the memory hierarchy, after which it retires and sets the forward progress bit to zero.

4 EVALUATION

4.1 Methodology

We use a benchmark suite that contains small programs to capture data structure access patterns along with high-level database operations such as TATP and TPCC. As we assume one TB dataset for 64 cores, each benchmark creates a dataset of 16GB for one core. We model the requests with an analytical Ziphian distribution chosen

such that the benchmarks trigger a DRAM miss every 5-25 μs on average.

We evaluate AstriFlash using QFlex, a cycle-accurate full-system simulator based on Flexus [44]. We assume an ARM Cortex-A57 core with a 2MB LLC. We scale down our 64-core requirements to one core for evaluation, and therefore model a DDR4 DRAM of 512MB in size, while the SSD [35] stores the 16GB dataset. Both DRAM and SSD use the traditional page size of 4KB.

4.2 Experimental Results

Throughput: In our evaluation, AstriFlash achieves up to 95% of DRAM-only system's throughput. The 3-5% throughput difference is because of the DRAM response wait, pipeline flush for every DRAM miss, and the scheduler overhead for switching threads. The OS-Swap configuration that we explained in Section 3 achieves 55% of the DRAM-only system's throughput as it has high overheads of page fault with each DRAM miss.

Service latency: AstriFlash achieves similar service latency distribution as the SSD-Sync (Sec. 3) because of our optimized priority-based scheduling policy. We also observe that AstriFlash with the priority scheduling (Sec. 3.2) exhibits 5x shorter service latency than AstriFlash without the scheduling mechanism as the scheduler keeps on executing new jobs even if the requested page has arrived for the pending job.

Tail latency: We study the latency distribution in detail for the TATP workload as it closely represents the short database operations present in modern datacenter workloads. AstriFlash has 16% degradation in 99-percentile latency compared to the DRAM-only system. The tail latency is worse because each DRAM miss in AstriFlash causes a ROB flush, and the following thread switch destroys the on-chip cache locality, thus incurring more on-chip misses. However, AstriFlash achieves a similar overall response latency distribution compared to the DRAM-only distribution. Even though the service latency is higher as it includes the SSD response time, the queueing latency becomes significantly lower.

5 CONCLUSION

In this paper, we propose AstriFlash, which is a new online flash-based memory hierarchy. AstriFlash eliminates the page fault overhead by employing a hardware/software co-designed switch-on-miss architecture. Our results show that AstriFlash serves data out of flash directly with 95% of the DRAM-only system's throughput while maintaining 99-percentile tail latency with only 16% degradation. Overall, AstriFlash reduces the memory cost by 20x, therefore providing a potential solution for future TB-scale memory systems.

REFERENCES

[1] Ahmed H. M. O. Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-Mei W. Hwu. 2019. 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)*, 971–985.

[2] AMD. 2019. AMD EPYC 7742. https://www.amd.com/en/products/cpu/amd-epyc-7742

[3] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1991. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 95–109.

[4] Duck-Ho Bae, Insoon Jo, Youra Choi, Joo Young Hwang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2018. 2B-SSD: The Case for Dual, Byte- and Block-Addressable Solid-State Drives.. In *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*, 425–438.

[5] Luiz André Barroso and Urs Hölzle. 2009. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.*

[6] Luiz André Barroso, Mike Marty, David A. Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

[7] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. 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)*, 49–65.

[8] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. 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)*, 107–120.

[9] Data Center Knowledge. 2017. Google Data Center FAQ. http://www.datacenterknowledge.com/archives/2017/03/16/google-data-center-faq/

[10] Francis M. David, Jeffrey C. Carlyle, and Roy H. Campbell. 2007. Context switch overheads for Linux on ARM platforms.. In *Experimental Computer Science*, 3.

[11] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[12] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim M. Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM footprint with NVM in facebook.. In *Proceedings of the 2018 EuroSys Conference*, 42:1–42:13.

[13] NVM Express. 2019. *NVM Express Base Specification v1.4.*

[14] Facebook. 2018. Facebook company info. https://newsroom.fb.com/company-info/

[15] Michael Ferdman, Almutaz Adileh, Yusuf Onur Koçberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. 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)*, 37–48.

[16] Gina Roos. 2018. DRAM and NAND Flash Prices Will Dive in Q1 2019. https://epsnews.com/2018/12/17/dram-nand-prices-dive/

[17] Intel. 2019. Intel® Xeon® Platinum 9282 Processor Delivers Leadership Performance . https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/performance-for-hpc-platforms-brief.pdf

[18] IO_URING. 2020. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf

[19] Bruce L. Jacob, Spencer W. Ng, and David T. Wang. 2008. *Memory Systems: Cache, DRAM, Disk.*

[20] Jeff Barr, AWS. 2019. 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/

[21] Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. 2014. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache.. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 25–37.

[22] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs.. In *8th USENIX Workshop on Hot Topics in Storage and File Systems, 2016.*

[23] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash ≈ Local Flash.. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)*, 345–359.

[24] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the performance of fast NVM storage with uDepot.. In *Proceedings of the 17th USENIX Conference on File and Storage Technologie (FAST)*, 1–15.

[25] Mohan Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselý, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. 2018. LATR: Lazy Translation Coherence.. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*, 651–664.

[26] Butler W. Lampson. 1983. Hints for Computer System Design.. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*, 33–48.

[27] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. 2019. 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)*, 603–616.

[28] Gabriel H. Loh and Mark D. Hill. 2011. 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)*, 454–464.

[29] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA.. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 3–18.

[30] Petros Koutoupis. 2019. Data in a Flash, Part IV: the Future of Memory Technologies. https://www.linuxjournal.com/content/data-flash-part-iv-future-memory-technologies

[31] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks.. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 325–341.

[32] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John K. Ousterhout. 2018. Arachne: Core-Aware Thread Management.. In *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*, 145–160.

[33] Moinuddin K. Qureshi and Gabriel H. Loh. 2012. 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)*, 235–246.

[34] CSG Rick Eads. 2019. PCI Express Gen 5 Specification and latest information of Gen 4 testing. https://www.keysight.com/upload/cmc_upload/All/PCI-Express5-Specification-and-latest-information-of-Gen4-testing.pdf

[35] Samsung. 2017. Optimizing Data Center Systems and Applications for Samsung SSDs. https://samsungsemiconductor-us.com/labs/pdfs/Samsung_Whitepaper_Ecosystem_v3.pdf

[36] SiFive. 2018. Manual. https://sifive.cdn.prismic.io/sifive%2Fac48c4fd-af85-46be-9dd9-22aa34ba6977_u54mc-core-complex-manual-v19.05.pdf

[37] Snehanshu Shah, Google Cloud. 2019. 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

[38] Avinash Sodani, Roger Gramunt, Jesús Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro* 36, 2 (2016), 34–46.

[39] SPDK. 2020. Storage Performance Development Kit. https://spdk.io/

[40] Dan Tsafrir. 2007. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops).. In *Experimental Computer Science*, 4.

[41] Dmitrii Ustiugov, Alexandros Daglis, Javier Picorel, Mark Sutherland, Edouard Bugnion, Babak Falsafi, and Dionisios N. Pnevmatikatos. 2018. Design guidelines for high-performance SCM hierarchies.. In *Proceedings of the International Symposium on Memory Systems (MemSys) 2018*, 3–16.

[42] Stavros Volos, Djordje Jevdjic, Babak Falsafi, and Boris Grot. 2017. Fat Caches for Scale-Out Servers. *IEEE Micro* 37, 2 (2017), 90–103.

[43] Frederick A. Ware, Javier Bueno, Liji Gopalakrishnan, Brent Haukness, Chris Haywood, Toni Juan, Eric Linstadt, Sally A. McKee, Steven C. Woo, Kenneth L. Wright, Craig Hampel, and Gary Bronner. 2018. Architecting a hardware-managed hybrid DIMM optimized for cost/performance.. In *Proceedings of the International Symposium on Memory Systems (MemSys) 2018*, 327–340.

[44] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. 2006. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro* 26, 4 (2006), 18–31.

[45] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. 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)*, 15–28.