Managing Memory Tiers with CXL in Virtualized Environments

Yuhong Zhong, *Columbia University, Microsoft Azure;* Daniel S. Berger, *Microsoft Azure, University of Washington;* Carl Waldspurger, *Carl Waldspurger Consulting;* Ryan Wee, *Columbia University;* Ishwar Agarwal, Rajat Agarwal, Frank Hady, and Karthik Kumar, *Intel;* Mark D. Hill, *University of Wisconsin–Madison;* Mosharaf Chowdhury, *University of Michigan;* Asaf Cidon, *Columbia University*

https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

Managing Memory Tiers with CXL in Virtualized Environments

Yuhong Zhong ♔ ⊞ Daniel S. Berger ⊞ W Carl Waldspurger* Ryan Wee ♔
Ishwar Agarwal i Rajat Agarwal i Frank Hady i Karthik Kumar i Mark D. Hill W
Mosharaf Chowdhury M Asaf Cidon ♔

♔ *Columbia University* ⊞ *Microsoft Azure* W *University of Washington* **Carl Waldspurger Consulting*
i *Intel* W *University of Wisconsin–Madison* M *University of Michigan*

Abstract

Cloud providers seek to deploy CXL-based memory to increase aggregate memory capacity, reduce costs, and lower carbon emissions. However, CXL accesses incur higher latency than local DRAM. Existing systems use software to manage data placement across memory tiers at page granularity. Cloud providers are reluctant to deploy software-based tiering due to high overheads in virtualized environments. Hardware-based memory tiering could place data at cacheline granularity, mitigating these drawbacks. However, hardware is oblivious to application-level performance.

We propose combining hardware-managed tiering with software-managed performance isolation to overcome the pitfalls of either approach. We introduce *Intel® Flat Memory Mode*, the first hardware-managed tiering system for CXL. Our evaluation on a full-system prototype demonstrates that it provides performance close to regular DRAM, with no more than 5% degradation for more than 82% of workloads. Despite such small slowdowns, we identify two challenges that can still degrade performance by up to 34% for "outlier" workloads: (1) memory contention across tenants, and (2) intra-tenant contention due to conflicting access patterns.

To address these challenges, we introduce *Memstrata*, a lightweight multi-tenant memory allocator. Memstrata employs page coloring to eliminate inter-VM contention. It improves performance for VMs with access patterns that are sensitive to hardware tiering by allocating them more local DRAM using an online slowdown estimator. In multi-VM experiments on prototype hardware, Memstrata is able to identify performance outliers and reduce their degradation from above 30% to below 6%, providing consistent performance across a wide range of workloads.

1 Introduction

Memory tiering is a promising approach to scale memory capacity and reduce the total cost of ownership (TCO) in datacenters. In public clouds, virtual machine (VM) memory sizes are increasing, with typical configurations of 4–32GB per virtual CPU [6, 7, 12]. However, the DRAM capacity accessible via DDR channels is lagging the rapid growth in available cores, due to physical limitations associated with

CPU core counts scaling faster than memory capacity

scaling the capacity of DDR DIMMs [73, 91, 92]. To this end, cloud providers are increasingly adding a *capacity memory tier* to augment regular locally-accessed DRAM, which we refer to as the *performance tier* [61, 72, 78, 84, 99].

The recent Compute Express Link (CXL) standard [8, 91] offers a new mechanism to access DRAM or non-volatile memory (NVM) over the PCIe bus, potentially expanding memory capacity significantly. In addition, CXL can reduce TCO and carbon emissions [83, 98] by provisioning it with decommissioned DRAM or NVM. This has led to broad investment in CXL memory by dozens of vendors [9, 10, 15, 25, 27, 28, 37]. The CXL standard envisions a variety of configurations. In this paper, we focus on the basic use case where a CXL memory device is locally attached and dedicated to a single host [91, 98]. This use case is deployable today, and extends to future memory pools [46, 77, 78].

Most prior work on memory tiering assumes software (e.g., the hypervisor or the OS) has full control over data placement, i.e., whether a particular page resides in the capacity tier or the performance tier [45, 61, 70, 74, 78, 84, 89, 90, 99, 100]. We term this *software-managed memory tiering*. Software-managed tiering needs to track memory accesses to identify frequently-accessed data to place in the performance tier. Since the hypervisor/OS is not involved in most memory accesses, it must rely on page table operations management (e.g., scanning access bits [61, 84, 100] or PTE poisoning [45, 70, 84]) or instruction sampling (e.g., Intel PEBS sampling [61, 74, 89] and AMD IBS [4]) to track memory accesses.

However, in our experience at Microsoft Azure, these approaches face severe limitations in virtualized environments (§2). For example, instruction sampling is not supported for VMs and has privacy implications. Fine-grained page table operations consume excessive host CPU cycles [44, 79]. In addition, with software-managed tiering, the hypervisor/OS can only manage memory at page granularity. This leads to suboptimal decisions [76] for the common case where a mix of hot and cold data resides on the same page. This is particularly problematic for hypervisors that use larger page sizes (e.g., 2 MB and 1 GB) to reduce overheads. All of these drawbacks make deploying software-based memory tiering techniques unattractive in general-purpose cloud environments.

This paper addresses these issues by introducing a hardware-managed memory tiering solution for CXL and a system that combines hardware-managed tiering with software-managed multi-tenant isolation. We introduce *Intel® Flat Memory Mode* as the first cache-line granular, hardware-managed memory tiering solution for CXL. Intel® Flat Memory Mode transparently manages data placement between the two tiers at cache-line granularity within the processor memory controller (MC). It exposes the aggregate capacity of both local DRAM and CXL memory to software by placing data *exclusively* at either of the tiers. The hardware promotes the most recently accessed lines to local DRAM by "swapping" them with the lines that used to occupy local DRAM.

To reduce the performance degradation of CXL memory, Intel® Flat Memory Mode supports a *mixed mode* which reserves a certain number of *dedicated* pages that are guaranteed to reside in local memory, while cache lines associated with the remaining pages may be placed in either local or CXL memory, based on whether they were recently accessed.

Intel® Flat Memory Mode should not be confused with the hardware-managed memory tiering solution for Intel® Optane™ NVDIMMs, known as *2LM* or *memory mode* [18, 65]. Such systems employ DRAM as an *inclusive* cache for non-volatile memory, which means the performance tier does not add capacity visible to software. The inclusive cache design makes them less useful for expanding memory capacity and reducing TCO. Additionally, 2LM only supports non-volatile memory, not CXL.

We describe Intel® Flat Memory Mode's design and evaluate it on a real CXL hardware prototype with a set of 115 workloads, comparing it to running fully on local DRAM. We find that 82% of workloads experience small (no more than 5%) slowdown in mixed mode. The remaining "outlier" workloads experience slowdowns up to 34%. We also observe that when VMs are co-located naïvely on the same server, they may interfere by "stealing" local DRAM from each other.

To address these challenges, we implement *Memstrata*, the first multi-tenant memory management software stack for hardware-managed tiered memory. Memstrata prevents inter-VM interference by identifying pages with conflicting cache lines, allocating them to the same VM using *page coloring*. In addition, Memstrata leverages a lightweight online slowdown estimator to assess the overhead incurred by tiered memory misses for each VM. It dynamically allocates dedicated local memory pages across VMs to improve the performance of those that are most sensitive to memory latency. Intel® Flat Memory Mode will be available in the Intel® Xeon® 6 Processor. We open source Memstrata at `https://bitbucket.org/yuhong_zhong/memstrata`.

We implement a full system prototype on a preproduction Intel® Xeon® 6 Processor that supports Intel® Flat Memory Mode. Memstrata is implemented within the Linux/KVM hypervisor and a new user-space management process. Our evaluation covers common workload and VM mixes observed in production at Azure. We find that Memstrata effectively prevents cross-VM interference and mitigates the tail in all scenarios. Specifically, the worst-case performance slowdown is reduced from 35% to less than 6% in realistic multi-VM experiments. Across all experiments, the maximum CPU overhead of Memstrata is 4% of a single core, which is less than 1% of a single core per VM.

We make the following contributions:

1. We introduce Intel® Flat Memory Mode, the first hardware-managed memory tiering mechanism for CXL. We evaluate it on a real CXL system, and show that for most applications it exhibits small slowdowns.
2. We design Memstrata, the first software multi-tenant management system for hardware-managed CXL that ensures performance isolation and minimizes VM slowdowns.
3. We study a wide range of workloads, and demonstrate that Intel® Flat Memory Mode combined with Memstrata eliminates almost all performance outliers, exhibiting minimal performance degradation compared to regular DRAM.

2 Background and Motivation

This section motivates hardware-managed memory tiering for CXL in virtualized environments.

2.1 Memory Tiering in Public Clouds

Current compute servers, which host customer VMs, use locally-attached DDR5 memory. With CPU core counts of 60-96 [66, 94] and Simultaneous Multithreading (SMT), achieving at least 4-8GB per virtual core requires 8-12 expensive dual-rank DIMMs (e.g., 64GB or 96GB). These DIMMs are the single biggest contributor to server cost [78, 99]. For large-memory VM sizes [6] or 128-288 core-count-CPUs [2, 14, 57], cloud providers need to use DIMMs with 3D stacking, which adds a multiplicative factor to per-GB memory cost [32]. Additionally, DIMMs make up 41% of a server's embodied carbon at Azure [49, 63, 83, 87, 98].

A second tier of memory can effectively reduce this cost. In modern servers, this second tier will use CXL [8, 91] to expand server memory capacity and bandwidth. This saves cost because cloud providers can use multiple smaller and cheaper DIMMs. Cost can be further reduced by *reusing memory from decomissioned servers*. Without CXL, DDR4 memory would be incompatible with modern servers. Instead of discarding DDR4 DIMMs, they can be repurposed for CXL memory. DDR4 reuse is supported today and has significant industry momentum [3, 26, 49, 83, 98]. This pattern can continue in future generations of DRAM, e.g., when DDR6 will be deployed, DDR5 can be reused with CXL. A third option is denser memory media [17, 38]. Both reusing old memory and using denser memory significantly cuts costs and carbon emissions. For example, attaching 40% of memory by reusing DDR4 can save over 20% of server embodied carbon emissions [83]. In this paper, we focus on this use case.

The downside of CXL memory is its latency overhead. CXL.mem customizes the PCIe link and transaction layers for low latency [91]. CPUs can natively access CXL memory via cacheable loads and stores, without involving page faults or DMAs. While an order of magnitude faster than RDMA, CXL is still slower than local DRAM as it essentially converts a parallel bus into a serial one. Depending on the specific memory controller, we measure that CXL memory has 2.02× the load-to-use latency of local DDR5 on the 5th Gen Intel® Xeon® Processor. A bidirectional ×8-CXL port at a typical 2:1 read:write-ratio matches a DDR5-4800 channel. In practice we use at least four ×8 ports.

2.2 Cloud Workload and Design Goals

Azure and other large cloud providers virtualize all workloads. VMs are generally small. For example, in a typical compute cluster at Azure, 40% of VMs use no more than two cores and 86% of VMs use no more than eight cores. Most modern hosts thus run dozens of VMs at any given time. Production cluster schedulers [47, 64, 97] increase utilization by mixing different workloads with no (or few) co-location constraints. Some constraints force similar workloads to be run across many hosts and racks, e.g., for fault tolerance. This leads to typical hosts running heterogeneous sets of workloads representing many different workload behaviors.

We derive the following four first-order design goals:

1. Compatibility with unmodified virtual machines. Do not assume guest cooperation.
2. Low host resource overheads. Cloud providers seek to sell almost all cores [44, 79]. Hosts typically use large 2 MB or 1 GB page sizes to reduce overhead.
3. No additional sources of cross-VM interference compared to running entirely on local memory.
4. Performance close to local memory for all workloads. Limit slowdown to about 5%, similar to prior work [78].

As observed in prior work, CXL slowdowns can be high for many workloads [78, 84]. This motivates managing data placement in tiered memory, either in software or in hardware.

2.3 Software-Managed Tiering

Software-managed tiering usually represents tiers as NUMA nodes [61, 78, 84]. Software explicitly allocates memory from a NUMA node and migrates pages between nodes. The hypervisor/OS typically tracks memory hotness to promote hot capacity-tier pages to the performance tier and demote cold pages to the capacity tier [61, 84, 89, 100]. Hotness tracking often relies on page table operations such as scanning PTE access bits [61, 84, 100] or temporarily unmapping entries to trigger minor page faults when they are accessed [45, 70, 84]. Other software tiering systems use instruction sampling (e.g., Intel PEBS [61, 74, 89] or AMD IBS [4]) to sample memory requests along with their associated memory addresses.

Problem 1: High host CPU cost. Tracking hotness at fine granularity is challenging in a cloud environment. Instruction sampling is typically unfeasible due to security and privacy concerns. Thus, cloud platforms need to rely on page-table-based approaches. We find that they can consume excessive host CPU cycles, which runs counter to design goal #2.

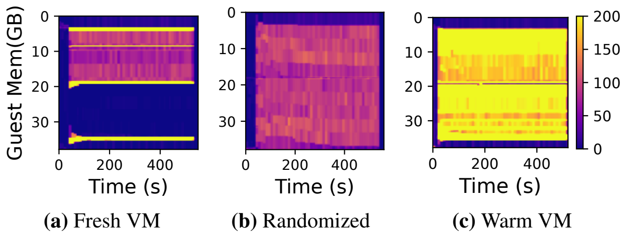


Figure 1: Memory access distribution of `bc-web` measured using DAMON in (a) a fresh VM, (b) a VM that enables free page randomization, and (c) a warm VM that has already run the workload 50 times. The right-hand y-axis represents the number of accesses captured by DAMON.

We measure the CPU overhead of TPP [84], a state-of-the-art software tiering system for guest kernels. We start a VM with 7.5 GB of local DRAM and 2.5 GB of second-tier memory. The VM runs YCSB A on FASTER [54], a production in-memory key-value store. FASTER consumes 8.3 GB memory in total, which means its memory cannot fit entirely in local DRAM. TPP devotes nearly an entire core to track memory accesses and migrate pages. This is caused by the frequent scanning of access bits in `kswapd`, which is used by TPP to demote cold pages. Without frequent access-bit scanning, TPP is unable to leave enough free space in local DRAM to promote hot pages. Scaling to larger systems and multiple VMs requires proportionally more CPU cycles.

The CPU overhead of page-table-based approaches can be reduced by exploiting spatial locality [11]. Unfortunately, spatial locality is limited in virtualized systems, which employ an additional layer of page table indirection. Additionally, a guest's free pages may be randomized for security [21]. We run the `bc-web` workload from the GAP benchmark suite [48] in a fresh VM, a VM with free page randomization enabled, and a warm VM that has already run the same workload 50 times. Figure 1 shows the memory access distribution measured using DAMON in the three VMs. While there is spatial locality in a fresh VM, locality disappears in both the VM with free page randomization and the warm VM. Approaches that scan guest page tables [90] may overcome fragmentation but run counter to goals #1 and #2.

Problem 2: Coarse-grained data placement. Software tiering moves entire pages, making a strong assumption about access locality. Many applications have spatially-sparse access patterns and thus perform poorly on software-managed tiering systems [53, 76]. Commonly, only a fraction of each page's cachelines are hot; moving such pages to the performance tier would be wasteful. This problem is exacerbated as cloud platforms use larger 2 MB and 1 GB page sizes to reduce page table depth and TLB misses [1, 34, 43, 45, 50].

To study how page size affects application performance, we

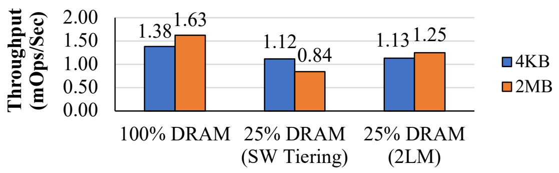


Figure 2: Throughput of FASTER on YCSB A with a varying DRAM ratio and different page sizes. The performance tier is DRAM and the second tier is Intel® Optane™ NVM.

run a VM with FASTER using the YCSB A workload. Since FASTER has a simple and predictable memory access pattern when running YCSB A, we analytically compute the popularity of each of its pages and always place the most popular pages in DRAM. Figure 2 shows that when FASTER's resident set can fit into DRAM, using 2 MB instead of 4 KB as the page size improves the throughput of FASTER by 18% thanks to the reduced page table depth and TLB misses. However, when only 25% of the resident set can fit into DRAM, a 2 MB page size degrades throughput by 25% due to the coarser data placement by software tiering. Google also reports that huge pages make cold page identification and demotion more challenging in their tiered memory production clusters [61].

2.4 Hardware-Managed Tiering

There are multiple variants of hardware memory tiering [68, 75, 85, 103]. They are typically implemented within the MC on the CPU SoC and behave similar to on-die CPU caches. Different memory tiers are typically invisible to software (no NUMA node) and fine-grained cache operations are visible only to the MC.

A well-known implementation of hardware tiering is "2LM" or "memory mode" for Intel® Optane™ NVDIMMs in the 2nd and the 3rd Gen Intel® Xeon® Scalable Processors [18, 65]. 2LM configures DRAM as a direct-mapped cache at cacheline granularity. It thus has no hotness tracking overhead and excels at managing workloads with limited locality [76], regardless of the page size (Figure 2). A major downside of using 2LM in the context of CXL is that the second tier is inclusive of the performance tier. This is wasteful, especially for the case of cloud providers who seek high performance and thus provision a large first memory tier. For example, provisioning 600 GB of DDR5 and 1000 GB of CXL memory means that only 1000 GB of overall memory is available, wasting 60% of CXL capacity.

3 Intel® Flat Memory Mode

In this section, we describe the hardware design of Intel® Flat Memory Mode and present a performance study on a wide range of applications.

3.1 Hardware Design

Intel® Flat Memory Mode overcomes the drawbacks of software-managed memory tiering by implementing the data placement within the MC. This allows it to manage data placement at cacheline granularity without involving host CPU. This design is especially useful in virtualized environments because the data placement is independent of the page size, and almost all the host CPU cores can be used to run VMs.

CXL memory ratio. To ensure minimal slowdown compared to local memory[1] (design goal #4), we assume a 1:1 ratio between the local memory and the CXL memory capacities. Other tiered memory deployments in industry also use small capacity-tier ratios to minimize slowdown: 33% at Meta [84] and 25% at Google [61]. With a 1:1 ratio, we can reduce the amount of local memory provisioned by 50%, which already significantly reduces memory cost. A higher CXL memory percentage may lead to higher slowdowns [78].

Exclusive placement. The amount of physical memory exposed to software is the aggregate capacity of both local DRAM and CXL memory. This is in contrast to 2LM, where the physical memory capacity is only as large as the size of the capacity tier (i.e., non-volatile memory). This design fully utilizes the capacity of both local DRAM and CXL memory by placing data *exclusively* at either of them, but not both. For example, once a cacheline is moved to local DRAM, it will no longer occupy any space in CXL memory.

Associativity. The associativity between physical memory and local memory is direct-mapped, which means each line in the physical memory address space can only be cached at one location in local memory. While direct-mapped associativity may lead to more conflict misses, this effect happens only after all the processor set-associative caches have missed. In addition, a straightforward implementation of set associativity would read multiple local DRAM lines to serve one main memory access, causing substantial bandwidth amplification.

Mixed mode. To further reduce local memory misses and improve the performance of workloads with cache-unfriendly memory access patterns, Intel® Flat Memory Mode supports adding dedicated local DRAM that is not hardware-tiered as a separate range in the physical memory address space. This dedicated local memory is exposed as a second NUMA node alongside the first NUMA node which contains the hardware-tiered memory. This dedicated NUMA node can be used for workloads suffering from severe local memory misses, as implemented in Memstrata (§4). We denote configurations where both hardware-tiered and dedicated NUMA nodes are present as *mixed mode*.

Mapping physical and local memory. In the hardware-tiered NUMA node, the ratio between local DRAM and the total physical memory capacity is 1:2. Thus, each line in local DRAM has 2 physical memory lines that map to it. This means that each 64 B line in the physical address space may be at one of two locations: either in local memory or in CXL memory. We use a modulo operation as the mapping function between the physical memory and local DRAM with the size of local DRAM (L GB) as the modulus. Figure 3 shows

[1]We use "local memory" and "local DRAM" interchangeably.

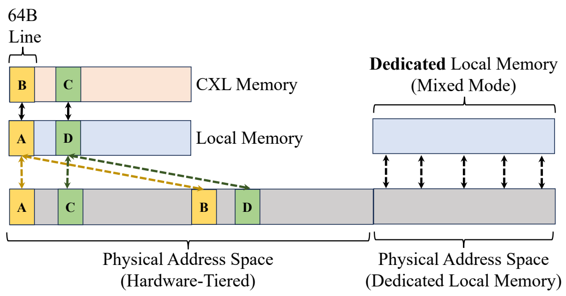
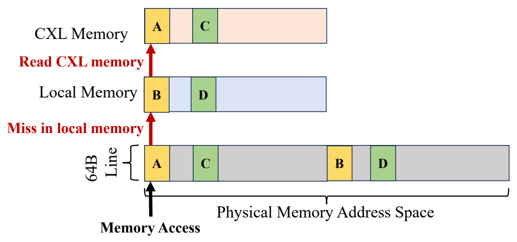
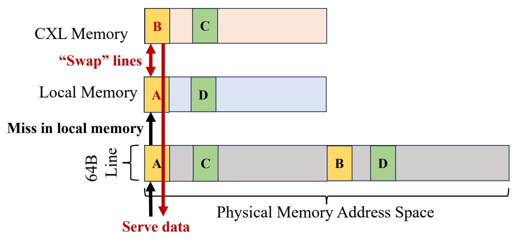

64B Line
B
C
CXL Memory
A
D
Local Memory
Dedicated Local Memory (Mixed Mode)
A
C
B
D
Physical Address Space (Hardware-Tiered)
Physical Address Space (Dedicated Local Memory)

Figure 3: Intel® Flat Memory Mode is a hardware tiering solution at cache line granularity between local and CXL memory. It supports a mixed mode option where part of the address space is not hardware managed and entirely backed by local memory for better performance. In the actively-managed region, cache lines A and B are mapped to the same local memory line, as are C and D. Only the most recently-accessed lines remain in local memory.



(a) A memory access to address A misses in local memory which leads to a subsequent read request to CXL memory.



(b) Once the MC receives the data from CXL memory, it will serve the data to the requesting core while swapping the two lines.

Figure 4: Main memory requests may miss in local memory. This triggers a cache line swap. Subsequent accesses to the same cache line hit local memory.

the mapping between the physical address space and local memory. This mapping halves the hardware-tiered physical memory, where the top half conflicts with the bottom half. For example, to hold physical memory addresses $[0\ GB, 1\ GB]$ in local DRAM, the hardware needs to evict $[L\ GB, L+1\ GB]$ to CXL memory.

Read and write operations. A memory access (i.e., Last Level Cache (LLC) miss) for a cache line in the hardware-tiered NUMA node may result in a "hit" or a "miss". The MC first reads local memory and determines if the requested line is in local memory. As only two physical memory lines can be cached at a given local memory line, the hardware only needs to maintain a single-bit tag to distinguish between them. If the tag matches the read request, the data is sent to the core that requested the line.

Otherwise, the requested line was a miss in local memory. Figure 4 shows how the MC handles a miss. The MC first fetches the data from CXL memory, and then sends the data to the core that requested the line. Meanwhile, the MC swaps the cache lines. Specifically, the MC writes the other line that used to occupy local memory to CXL memory. The MC writes the newly requested line to local memory.

When the MC receives a write request, just like the read flow, it needs to first locate and read the line into the processor caches. When the write is evicted from the processor caches (since writes are posted), the data is written to local memory.

Request interleaving. To achieve maximum bandwidth, we interleave local memory requests across memory channels and interleave CXL memory requests across CXL devices at cache line granularity within the same NUMA node.

3.2 Application Performance

Adding CXL memory capacity can provide clear performance benefits to memory-hungry applications, due to reduced paging to disk, and higher page-cache hit rates. However, these benefits depends heavily on the specific workloads and the total amount of memory available to them. To conservatively evaluate Intel® Flat Memory Mode, we compare it with X local memory and Y CXL memory to a baseline configured with $X+Y$ local memory. We evaluate the performance using a wide range of applications on a prototype CPU that supports Intel® Flat Memory Mode. The detailed hardware setup is described in §6. We use 115 workloads in total, including:

- **Web**: DaCapo [51], Renaissance [88], Ruby YJIT [39], and DeathStarBench benchmarks [62]
- **Database**: TPC-C [41] on Silo [96] and TPC-H [42] on PostgreSQL [33]
- **Machine learning (ML)**: DLRM benchmark [67, 86]
- **Key-value (KV) store**: YCSB [58] on FASTER [54], Redis [36], and memcached [23]
- **Big data**: HiBench [13] on Spark [104]
- **Graph processing**: GAP benchmark [48]
- **Scientific computing**: SPEC CPU 2017 [40]

We measure the performance of each workload running inside a VM on Linux/KVM. The VM memory size is chosen by rounding up the workload's peak resident set size to the nearest common VM memory size on public cloud platforms (2 GB, 4 GB, 8 GB, 16 GB, 32 GB, and 64 GB) [6, 7, 12]. We run each workload in four different settings: (1) local DRAM only, (2) CXL memory only, (3) hardware-tiered memory only, and (4) a mixed mode with 33% dedicated local DRAM and 67% hardware-tiered memory.

When allocating hardware-tiered memory pages to a VM, we allocate pairs of conflicting pages so that half of the allocation can be cached in local DRAM. As a result, our mixed mode configuration consists of 67% local DRAM (33% dedicated + half of 67% hardware-tiered) and 33% CXL memory. We conservatively choose 67% as the percentage of hardware-tiered memory as this configuration can already reduce the

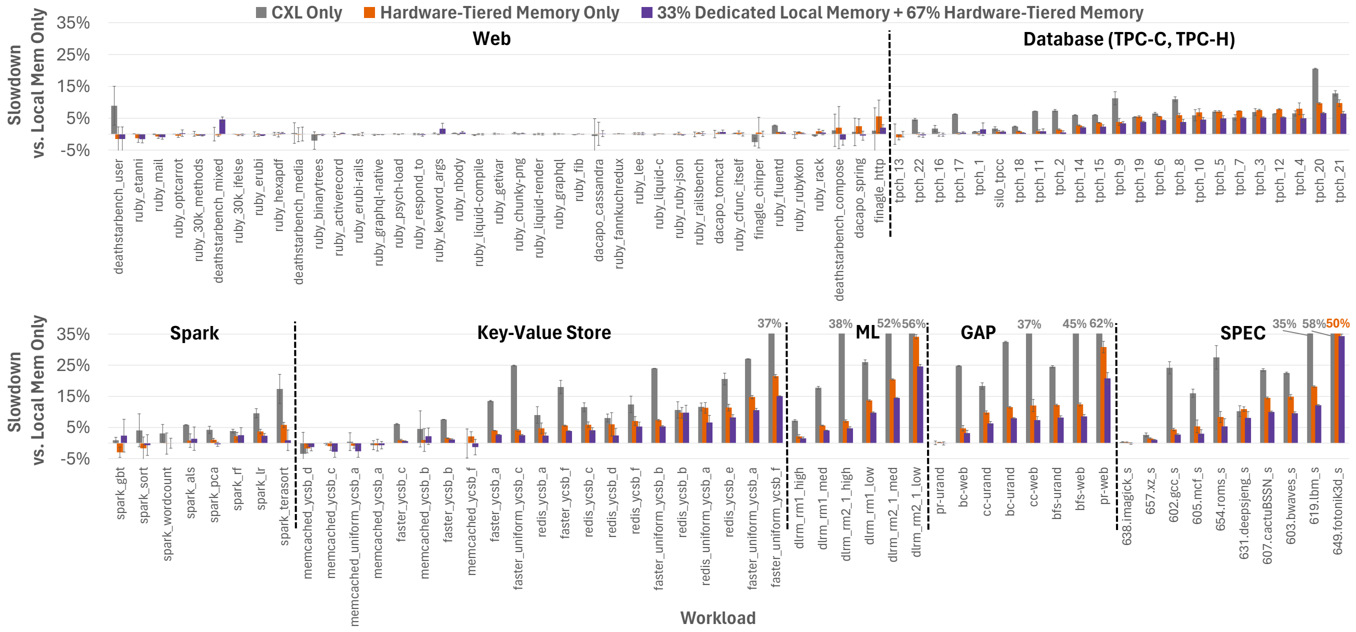


Figure 5: Slowdowns of 115 workloads when using only CXL memory, 100% hardware-tiered memory, or a mixed mode with 33% dedicated memory and 67% hardware-tiered memory. The error bars represent the standard deviations of slowdowns across three runs.

provisioned local memory by 33%. We randomly assign the dedicated local pages across a VM's address space. Each VM sees a uniform address space, as this is the default configuration on cloud platforms.

Figure 5 presents the results, categorizing workloads by their types, and ordering them by their relative slowdowns of hardware tiering. We refer to applications that experience slowdowns above 5% (see goal #4 in §2.2) as *outliers*. Web workloads experience negligible slowdowns even with CXL memory only, indicating their insensitivity to main memory latency. In contrast, database and Spark workloads have some outliers with slowdowns of up to 20% when using only CXL memory. Hardware tiering reduces the slowdowns for most outliers to close to or lower than 5%, although a few outliers still exhibit around 10% degradation. Other categories have more outliers with CXL memory only, with slowdowns of up to 58%. Hardware tiering significantly alleviates the performance degradation for these outliers. For example, the slowdown of FASTER with uniform YCSB C is reduced from 25% to 4%. However, even with reduced degradation, some outliers still experience slowdowns of up to 50% with hardware-tiered memory due to cache-unfriendly memory access patterns and the associated high local memory miss ratios. The most severe outlier, `649.fotonik3d_s`, suffers from a 41% miss ratio due to its large working set and scan-like memory access pattern.

The mixed mode with 33% dedicated local memory improves the performance of these outliers thanks to the reduction in the number of pages that conflict on local DRAM. Overall, with only hardware-tiered memory, 73% of the workloads experience no more than 5% slowdown, and 86% experience no more than 10% slowdown. In the mixed mode, the percentage of workloads with no more than 5% slowdown increases to 82%, and 95% of the workloads experience no more than 10% slowdown. These results are encouraging: despite the non-negligible slowdown of CXL compared to local DRAM, most applications have small slowdowns in the mixed mode. However, even in the mixed mode, some applications experience non-trivial degradation of up to 34%. This observation motivates the use of software to dynamically allocate dedicated memory pages across VMs to consistently achieve minimal slowdown, because the hardware is oblivious to which VMs suffer from local memory misses.

3.3 Noisy Neighbors

In Intel® Flat Memory Mode, two conflicting physical memory lines compete for the same local DRAM line, and only the most recently accessed one can be cached in local DRAM. Therefore, when conflicting pages are allocated to different VMs, they may contend for local memory, resulting in performance interference.

We study this inter-VM interference due to local DRAM conflicts by running two VMs: a normal VM and a *noisy neighbor* VM. In the normal VM, we run one of the workloads from our workload set, while in the noisy-neighbor VM, we always run a 6-thread Intel® Memory Latency Checker (MLC) [19], which scans its memory in a busy loop. We always scale MLC to have the same memory size as the normal VM. We configure MLC to use only 6 threads so that neither the local DRAM bandwidth nor the CXL memory bandwidth is saturated. Running MLC as the workload in the noisy neighbor VM allows us to estimate the worst-case interference, as MLC is optimized to be memory intensive.

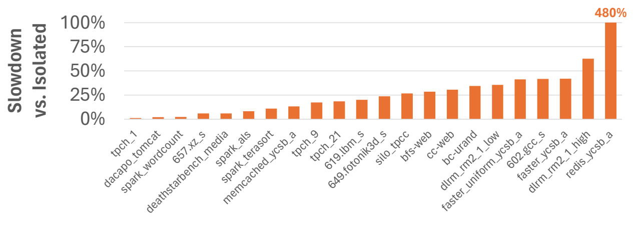


Figure 6: Slowdown caused by the noisy neighbor due to local DRAM conflicts.

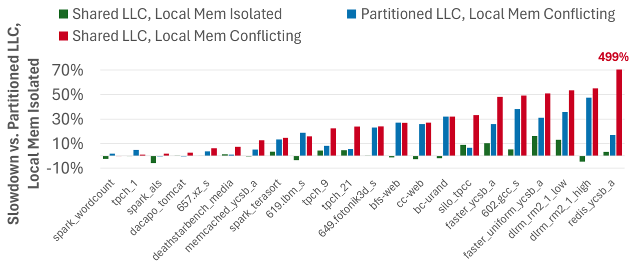


Figure 7: Inter-VM interference caused by LLC contention and local DRAM contention.

The experiments are conducted in two settings:

1. **Isolated.** Allocate conflicting pages to the same VM to ensure each VM only conflicts with itself.
2. **Conflicting.** Allocate conflicting pages to different VMs. This setting measures the worst-case interference since the noisy neighbor VM might monopolize local DRAM.

Figure 6 shows the slowdown of conflicting compared to isolated with a sampled set of representative workloads from each category. 73% of the workloads experience more than 10% slowdown because of local DRAM conflicts. The massive slowdown of Redis is because we use p95 latency as its performance metric. Redis always has some requests with extreme latency (4× higher than the median), and the contention causes the percentage of these requests to exceed 5%, which translates to a 480% slowdown in p95 latency. The results indicate that without any software management to isolate local DRAM conflicts, VMs running on the same host could cause significant performance interference to each other.

Besides local DRAM contention, other sources of interference in multi-tenant environments include contention in the LLC and power. We also study how LLC contention compares to local memory contention in Intel® Flat Memory Mode. We again use a normal VM and a noisy neighbor VM to measure interference. Besides configuring how the two VMs conflict with each other in local memory, we configure the LLC in two settings: (1) sharing LLC across two VMs, or (2) partitioning LLC evenly between two VMs. We use Intel's Cache Allocation Technology [20] to partition the LLC.

When the LLC is partitioned and local DRAM conflicts are isolated, there will be no interference caused by either LLC or local memory contention. When the LLC is shared but local DRAM conflicts are still isolated, we will only observe LLC interference. Similarly, we can only observe local memory interference if the LLC is partitioned and the two VMs are conflicting in local DRAM. Finally, to measure both LLC and local memory interference, we can share the LLC and also let the two VMs are conflicting in local memory.

Figure 7 shows the slowdowns caused by either LLC or local memory interference, as well as the slowdowns when both types of interference exist. Compared to LLC interference, local memory interference is typically larger. In addition, the workloads that suffer from LLC interference also suffer from local memory interference. When both LLC and local memory interference exist, those workloads experience higher slowdowns than when there is a single source of interference. These results again indicate that we should isolate local DRAM conflicts to achieve design goal #3.

4 Memstrata

Memstrata leaves the heavy lifting of fine-grained memory management to the hardware-managed tiering layer at the MC. It provides consistent performance by integrating a lightweight software stack with the virtualization host. This achieves the first two design goals (§2.2).

To provide performance isolation (design goal #3), Memstrata adapts page coloring [69, 105], a classic technique for partitioning CPU caches, to the CXL setting. Memstrata identifies all conflicting pages and allocates conflicting pairs to the same VM, ensuring no inter-VM conflicts (§4.1).

To improve the performance of outliers (design goal #4), Memstrata dynamically allocates dedicated local memory pages across VMs to reduce the outliers' local memory miss rates. Our key insight is that many workloads exhibit low slowdowns even without any dedicated local memory. Therefore, if the hypervisor can identify outlier VMs and move dedicated local memory pages to them, it can limit their slowdowns.

However, the hypervisor has limited visibility into the workloads running inside VMs, making it challenging to identify outliers. Although monitoring local DRAM miss rates seems attractive for detecting outliers, we cannot directly measure per-core or per-VM miss rates because data placement is implemented in the MC, so hardware performance counters can report only the system-wide local memory miss rate.

To tackle these challenges, we analyze numerous performance events measured during our application performance study (§3.2) and propose a proxy to estimate per-VM miss rates. By combining the estimated miss rate with other performance metrics, we can accurately predict the slowdown of a VM using a simple online ML model (§4.2). This model is used by a dynamic page allocator to migrate dedicated local memory pages across VMs, minimizing slowdowns across all workloads with negligible CPU overhead (§4.3). Figure 8 shows an overview and the workflow of Memstrata.

4.1 Page Coloring

Page coloring is a software technique that has been widely used to partition shared processor caches (e.g., the LLC) in a modern CPU [80, 93, 95, 102, 105]. CPU caches are commonly

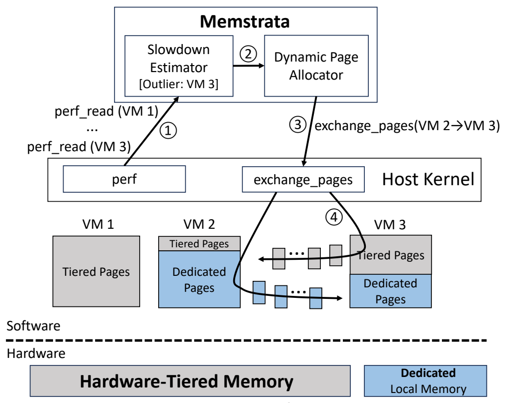


Figure 8: Overview of Memstrata. ①: Memstrata reads the performance events of each VM and runs the slowdown estimator. ②: Slowdown estimations are used to decide the allocation of dedicated pages. ③: The dynamic page allocator uses the `exchange_pages` syscall to migrate dedicated pages to an outlier. ④: The hardware-tiered pages of the outlier are exchanged with the dedicated pages of a non-outlier VM.

organized in a set-associative (or direct-mapped) manner, in which each physical memory address is mapped to the index of a single set in the cache. If the indexing function is known, then software can determine the subset of cache indices associated with a given memory page, referred to as its *page color*. Since main memory is much larger than the cache, many memory pages have the same color, which means that they compete for the same limited cache space.

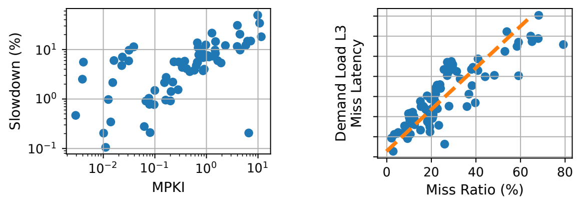
System software can control the amount of cache space that may be used by different applications by allocating pages to them with particular colors. For example, a hypervisor can allocate host-physical pages so that each VM uses distinct colors that are disjoint from other VMs.

Similar to shared processor caches, in Intel® Flat Memory Mode, local memory is shared among all VMs within the same NUMA node.[2] Physical memory lines that are mapped to the same local memory line compete for the same local memory space, which will contain the one accessed most recently. Therefore, we can adopt page coloring to partition local memory pages across different VMs to avoid inter-VM local memory conflicts.

We implement page coloring in the context of a virtualized system configured with Intel® Flat Memory Mode. The implementation consists of changing the free-page management logic and the page allocator in the host Linux kernel. We modify Linux's free-page management to group the physical pages that map to the same local DRAM page.

To avoid performance interference due to inter-VM conflicts, the page allocator always allocates the physical pages that map to the same local memory page to the same VM. This isolates each VM, ensuring that it can conflict only with

[2]For clarity, we only discuss the hardware-tiered memory *without* dedicated local memory in this subsection. Dedicated local memory is contained in a distinct NUMA node from the local memory associated with the hardware-tiered memory.



(a) MPKI vs. slowdown **(b)** Miss ratio vs. L3 miss latency

Figure 9: Correlation between performance metrics.

itself, eliminating the possibility of conflicts with other VMs. Isolating local memory conflicts within VMs also prevents their use as inter-VM side channels, similar to those exploited in other caching systems to leak memory access patterns and to exfiltrate data across trust boundaries [71, 81, 82].

While one might expect this to cause poor performance, there are, in fact, many cold or unused cache lines. This means that we actually observe relatively low local memory miss rates (§3.2).

4.2 Identifying Outliers

To improve the performance of outlier workloads, we first need to determine which VMs suffer from high slowdowns because of local DRAM misses. Since the workloads running inside VMs are opaque to the hypervisor, we cannot rely on application-level performance metrics for outlier detection.

Fortunately, modern processors provide performance counters that can be used to infer VM performance characteristics with low overhead [5, 16, 31, 35, 101]. Our prototype CPU also supports various performance events. In our large-scale performance study of Intel® Flat Memory Mode using 115 workloads (§3.2), we configured the CPU to count all performance events for each workload by time-multiplexing them, yielding 151 performance metrics based on raw event counts.

MPKI and its proxy. Among all the performance metrics, local memory miss rate seems promising for detecting outliers, since the local memory is treated as a cache. Specifically, we examine MPKI, which measures the number of local memory misses per thousand instructions. Figure 9a shows that MPKI is correlated ($r^2 = 0.73$) with application slowdown.

Estimating slowdown for a single VM requires per-VM miss rates. As each VM is pinned to a set of exclusive CPU cores, this suggests aggregating the miss rates across its associated cores to compute the VM-level miss rate. Unfortunately, since cacheline promotions and demotions are handled in the MC, it is not easy to track per-core misses, and the system-wide miss rate is insufficient for outlier detection. Although future hardware may support measuring per-core miss rates, this is not implemented in the current prototype.

To work around this limitation, we analyze other performance metrics that can be tracked with per-core granularity to find a proxy for the per-core miss rate. As shown in Figure 9b, we find that the L3 miss latency[3] of demand loads event has a strong linear correlation ($r^2 = 0.87$) with the local

[3]Figure 9b omits the y-axis latency scale for confidentiality.

memory miss ratio (not MPKI), defined as the percentage of main memory requests that miss in local memory. This is not surprising, as Intel® Flat Memory Mode exhibits stable hit and miss latencies unless the memory bandwidth is saturated. We leverage this observation by fitting a linear model to estimate the local memory miss ratio from demand load L3 miss latency. The estimated miss ratio is translated to MPKI by multiplying it with the main memory request count and then dividing by the instruction count. We use this estimated MPKI as a proxy for the actual MPKI.

A limitation of this approach is that demand loads represent only a portion of main memory requests. Other sources include read-for-ownership (RFO) requests, non-temporal stores, CPU cache writebacks, and CPU cache prefetches. However, we find that the estimated MPKI works well in practice. When combined with other metrics, it can be used to predict VM slowdown accurately.

Using a model to detect outliers. Although MPKI strongly correlates with application slowdown, we find that MPKI alone is not sufficient to identify outliers because applications can have different sensitivities to memory latency. Therefore we use an online random forest binary classifier [52] to determine whether a VM will experience more than 5% slowdown. We choose a random forest classifier because it performs well with low-level performance metrics [78], is lightweight, and does not require a GPU. The input to the classifier includes the estimated MPKI along with four additional per-VM metrics that also exhibit useful correlations, selected by computing the relative importance of features during classifier training: (a) L3 miss latency of demand loads, (b) L2 miss latency of demand loads, (c) data TLB load miss latency, and (d) L2 MPKI of demand loads. We evenly split the workloads into training and validation sets, and configure the random forest with 100 decision-tree estimators. The classifier achieves 100% accuracy on the training set and 88% accuracy on the validation set, demonstrating the ability to detect outliers across a diverse set of workloads. In contrast, using MPKI as the only feature achieves only 63% accuracy on the validation set.

4.3 Dynamic Page Allocator

The Memstrata dynamic page allocator manages how dedicated local memory pages are allocated across VMs. It uses the ML model to detect outlier VMs, and migrates dedicated local memory pages accordingly to achieve minimum slowdown across all workloads. Within each VM, the page allocator assigns dedicated local memory pages to guest physical pages randomly. We also implemented an alternative hotness-based approach that prioritizes popular guest physical pages, but found that its overhead typically exceeds its benefit.

Inter-VM page allocation. The dynamic page allocator allocates dedicated pages to each VM based on its performance events and slowdown predictions from the ML model. It starts by measuring the events needed by the ML model for a given time interval (10 seconds, by default), and runs the model

```
def comparator(vm1, vm2):
  if vm1.isOutlier != vm2.isOutlier:
    return vm2.isOutlier
  return vm1.avgMissCount < vm2.avgMissCount

def migrate(vms, timeInterval, ewma, stepRatio):
  while systemIsRunning():
    sleep(timeInterval)
    updatePerfMetrics(vms, ewma)
    predictSlowdown(vms)
    sort(vms, comparator)
    donor = 0
    for borrower in range(len(vms) - 1, 0, -1):
      if not vms[borrower].isOutlier:
        break
      toBorrow = vms[borrower].pages * stepRatio
      while donor < borrower and toBorrow > 0:
        toDonate = (vms[donor].pages * stepRatio
                    - vms[donor].donated)
        toMigrate = min(toBorrow, toDonate)
        doMigrate(borrower, donor, toMigrate)
        toBorrow -= toMigrate
        vms[donor].donated += toMigrate
        if toDonate == toMigrate:
          donor += 1
```

Listing 1: Inter-VM page migration algorithm.

to predict if the slowdown for each VM is greater than 5%. The 10-second interval enables the page allocator to react quickly to changes, while averaging out noise associated with low-level event counts. To reduce the effect of short-term variations, we employ an exponentially weighted moving average (EWMA) to smooth the performance metrics derived from the event counts (EWMA constant $\alpha = 0.2$, by default).

Once the page allocator obtains the performance metrics and slowdown prediction for each VM, it decides how dedicated local memory pages should be migrated across VMs. Listing 1 presents the page migration algorithm. The page allocator first ranks the VMs based on their predicted slowdowns and the average number of local DRAM misses per allocated hardware-tiered page. The average miss count is computed by multiplying the estimated miss ratio with the main memory request count, and dividing the result by the number of hardware-tiered pages assigned to the VM. The VMs predicted to have less than 5% slowdown receive lower ranks than the outlier VMs. VMs that have the same slowdown prediction are ordered based on their average miss count. The intuition is that prioritizing VMs with higher average miss counts minimizes system-wide local memory misses, since they benefit more from a fixed amount of dedicated local memory pages compared to others [56].

After the VMs are sorted according to their ranks, the allocator repeatedly migrates dedicated pages from the VM with the lowest rank to the VM with the highest rank. To prevent large performance fluctuations, it never migrates more than a fraction `stepRatio` of each VM's pages (10%, by default) during each step. Only VMs predicted to be outliers can receive dedicated local memory pages from others.

To migrate dedicated local memory pages from VM 1 to

VM 2, the page allocator first selects a given number of dedicated pages from VM 1 and the same amount of hardware-tiered pages from VM 2. It then exchanges the selected pages between the two VMs. To avoid introducing inter-VM local DRAM conflicts, conflicting pages are always migrated together. After one round of page migrations, the page allocator stops migrating and measures performance events over the next `timeInterval` before the next round of migration.

Launching and terminating VMs. In cloud environments, running VMs may be terminated and new VMs may be launched at any time. To avoid disruptions, the page allocator first removes a terminating VM from the list of active VMs that participate in page migration. The terminating VM can then be shut down, causing its pages to be returned to the free page pool maintained by the host kernel.

When a new VM is launched, its initial allocation consists of existing free pages from the host kernel, which could be any mix of hardware-tiered pages and dedicated local memory pages. Since the dynamic page allocator does not have any prior information about the new VM, once it is added to the active VM list, the allocator migrates pages so that the new VM contains the same percentage of dedicated pages as the entire system. For example, if the overall system has a total of 33% dedicated local memory and 67% hardware-tiered memory, the new VM will also have 33% dedicated pages. This initial migration is performed by taking (or giving) dedicated pages to (or from) other existing VMs, each of which contributes (or receives) the same number of dedicated pages.

Assigning dedicated memory. By default, the dynamic page allocator assigns dedicated local memory pages to guest physical pages randomly within each VM. We also experimented with an alternative hotness-based page allocation option, which prioritizes popular guest physical pages when allocating dedicated pages. To identify popular guest physical pages, we employ DAMON [11], a low-overhead memory access tracking subsystem integrated into the mainstream Linux kernel. We use DAMON's default settings, but configure its aggregation period to match the `timeInterval` used by the allocator. After finishing inter-VM page migration, the allocator checks the per-region access counts reported by DAMON. Using simple thresholds (cold = 0, hot $\geq$ 20, by default), it exchanges any hardware-tiered pages in hot regions with dedicated pages in cold regions. To avoid large performance fluctuations, such intra-VM migrations are limited to a small fraction (2%, by default) of the VM memory size.

However, we found that the overhead of this hotness-based approach exceeds it benefit (§6.2). Similar to software-managed tiering, it consumes significant CPU cycles to track memory accesses (§2.3). Therefore, by default, Memstrata simply assigns dedicated local pages randomly.

5 Memstrata Implementation

Memstrata's implementation consists of implementing page coloring and the page-exchange system call in the host Linux kernel (v5.19, 2729 LOC), modifying QEMU (v6.2, 60 LOC) to preallocate guest memory for VMs, and building the main functionality of Memstrata as a privileged userspace process that runs on the host (2190 LOC, C++). Like QEMU, Memstrata uses 2 MB as the page size at the host level.

The page-exchange system call `exchange_pages(pid_1, pid_2, page_arr_1, page_arr_2, num_pages)` accepts the PIDs of two processes, an array of linear addresses for each process, and the number of pages to exchange. One can exchange pages within a single process by specifying the same value for `pid_1` and `pid_2`. The syscall is implemented in the host kernel based on the `migrate_pages()` function. To exchange two physical pages, the kernel initially moves the first page to a temporary physical page, then transfers the second page to occupy the first page's original location, and finally relocates the temporary page to the second page's initial position. It uses a Linux MMU notifier [24] to synchronize the secondary page table used by the VM and the QEMU host-level page table.

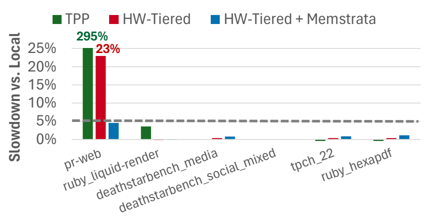
We implement only the necessary mechanisms (i.e., page coloring and page exchange) in the host kernel and run Memstrata as a privileged userspace process, facilitating debugging and extensions. The userspace process configures and reads performance events via the `perf` interface exposed by the host Linux kernel. It uses the custom page-exchange syscall to exchange pages between two VMs, or within a single VM. We use ONNX [60] to run the ML model in the userspace process. To synchronize Memstrata with VM launching and termination, we use POSIX message queues to let the VM scheduler communicate with Memstrata.

6 Evaluation

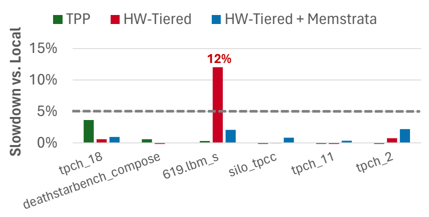
In this section we seek to answer the following questions:

1. How does Intel® Flat Memory Mode compare to software-managed tiering? (§6.1)
2. Can Memstrata improve the performance of outliers without impacting other applications? (§6.1)
3. How does dedicated memory page allocation affect application performance? (§6.2)
4. Is Memstrata sensitive to its parameters? (§6.3)

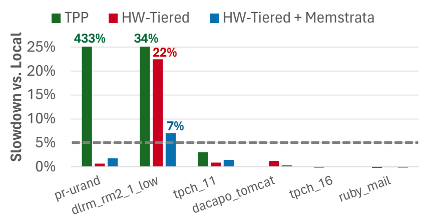
Experimental setup. We use a pre-production Intel® Xeon® 6 Processor that implements Intel® Flat Memory Mode. Our test server contains a single socket with 128GB DDR5 local memory and 128GB DDR5 CXL memory. The CXL memory is attached via three CXL cards, which each hold two DDR5-4800 DIMMs and offer an x16 PCIe5 CXL connection. We use a preproduction Astera Labs Leo CXL Smart Memory Controller [22]. The idle latency of the CXL memory is roughly 200-220% the latency of the local memory, and the max bandwidth per CXL card is around 50 GB/s [91]. Although we use DDR5-4800 DIMMs in the CXL cards, we believe the results are transferable to DDR4 DIMMs because the actual CXL bandwidth usage is always below the limit



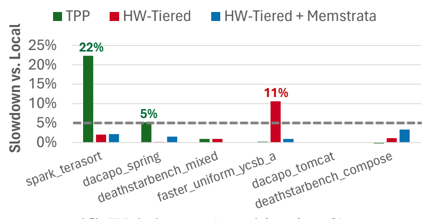
(a) Web-heavy (combination 1)



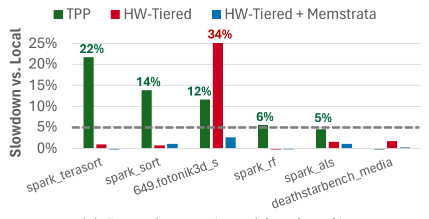
(b) Data-heavy (combination 1)



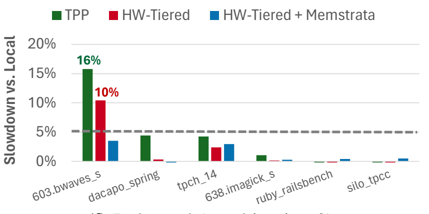
(c) Balanced (combination 1)



(d) Web-heavy (combination 2)



(e) Data-heavy (combination 2)



(f) Balanced (combination 2)

Figure 10: Application slowdown of TPP and hardware tiering with and without Memstrata using different workload combinations.

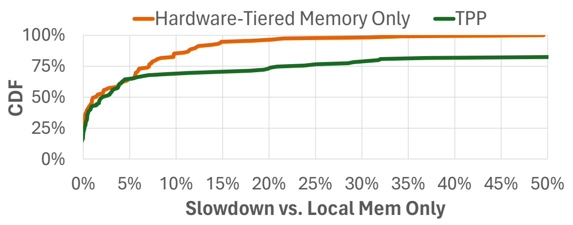


Figure 11: Slowdown distribution of hardware-tiered memory and TPP. TPP is configured with 50% local memory to match the local memory ratio of hardware-tiered memory.

of DDR4 DIMMs. Since the CPU is pre-production, the core count, frequency, and CPU cache sizes may not reflect those of the final product. For confidentiality, we cannot share the detailed technical specification. Similarly, these CXL cards are pre-production and future versions may be faster.

We focus on end-to-end application performance to demonstrate the performance benefits of Memstrata. Our software stack comprises Ubuntu 22.04, modified Linux 5.19, and QEMU/KVM 6.2. Hyperthreading and CPU frequency scaling are disabled. We pin each VM's virtual cores to physical cores in a 1:1 manner. We set the VM memory size of each workload by rounding up its peak resident set size to the next largest VM memory size offered on public cloud platforms.

We do not use public or Azure VM traces since they do not label workloads for VMs. This is because public cloud providers are not generally aware of workloads running inside VMs. Therefore, we rely on analyses of the composition of 188 internal workloads over 100,000 VMs at Azure [98], which reveal that web (31%), big data (32%), and ML (11%) workloads constitute most of the VMs. The remaining workload categories include DevOps and real-time communication workloads, which are challenging to run and have few open-source representatives. Therefore, we focus on the web, big data, and ML workload categories and reuse the set of workloads from §3.2 to match this composition. We exclude the workloads that have unstable performance. With the prototype system only offering 128 GB local memory, we also exclude workloads that require more than 32 GB of memory, so that we can measure a multi-VM local-only baseline.

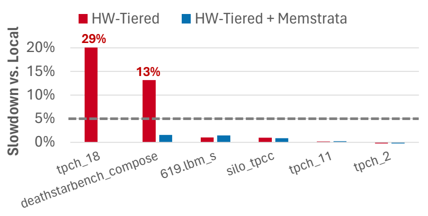
We compare Intel® Flat Memory Mode without and with Memstrata (referred to in the figures as "HW-Tiered" and "HW-Tiered + Memstrata", respectively). To emulate a setting without Memstrata, we use a static allocation scheme in which the percentage of dedicated pages in each VM remains constant over time. In contrast, Memstrata dynamically migrates dedicated pages across VMs to minimize slowdown. All settings use page coloring to avoid inter-VM conflicts.

We also compare hardware tiering without Memstrata to TPP [84], a state-of-the-art software-tiering approach. Since TPP does not support virtualization, we run it within each one of the isolated VMs, with a 2:1 ratio of local DRAM to CXL memory, matching the default setting of hardware tiering. Similar to hardware tiering without Memstrata, TPP does not move memory across VMs. The open-sourced TPP has an issue that wastes some local memory because it allocates local memory only from the `NORMAL` memory zone [30]. We have fixed this issue to enable TPP to perform better.

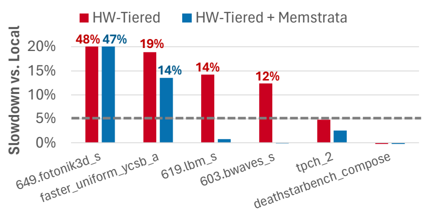
6.1 Performance Benefits

We assume a workload mix with about $\frac{1}{6}$ of workloads being outliers with hardware tiering, as our results show that 20% of the web, big data, and ML workloads experience more than 5% slowdown with hardware tiering (§3.2). As our prototype offers only 128 GB local memory, we must scale down the set of workloads typical for a large server. We scale to six VMs, typically with a single outlier workload[4]. We also consider the less likely scenarios of 2/6 and 4/6 outliers, as well as

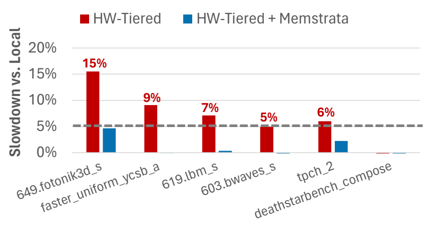
[4]If compute servers indeed were to only host six VMs, scenarios with multiple outliers would be common. However, we seek to represent a scaled-down typical compute server with large VM counts (§2.2). Due to large-number effects most servers will thus have a $\frac{1}{6}$ ratio of outlier workloads. One can also integrate our slowdown estimator (§4.2) into the VM scheduler to explicitly prevent colocating many outliers (§7).



(a) 2 outliers with 33% dedicated local memory



(b) 4 outliers with 33% dedicated local memory



(c) 4 outliers with 67% dedicated local memory

Figure 12: Application slowdown when the workload combination contains 2 or 4 outliers.

dynamic VM arrivals. We start each VM with 33% dedicated local memory pages and 67% hardware-tiered pages.

Common cases. As web and data workloads are the dominant workload categories at Azure [98], we choose the following workload mixes to evaluate Memstrata:

1. **Web-heavy**: 4 web, 1 data, and 1 outlier.
2. **Data-heavy**: 1 web, 4 data, and 1 outlier.
3. **Balanced**: 2 web, 2 data, 1 others, and 1 outlier.

For each workload mix, we generate two workload combinations from our workload set. Each combination starts the six workloads simultaneously at the beginning.

Figure 10 shows the slowdown across the entire run. Under TPP, there are significantly more outliers than with hardware tiering, despite TPP having an unfair advantage: TPP has visibility inside the VM, and knows which pages are being used at the 4 KB granularity. Consequently, TPP can place the entire working set into local DRAM if its size does not exceed the local DRAM size. Such visibility assumes guest cooperation and is not compatible with design goal #1 (§2.2).

Interestingly, TPP and hardware tiering sometimes have different outliers. We observe that TPP achieves minimal slowdown whenever the working set can fit into local memory (e.g., SPEC's `619.lbm_s` and FASTER with uniform YCSB A), which is the target use case of TPP [84]. However, if the working set is too large, TPP experiences severe thrashing, causing massive TLB invalidations and page faults due to frequent page migration. For example, in an extreme case where TPP causes `pr-web` to have a 295% slowdown (Figure 10a), TPP migrates memory at 22 GB/s in a 32 GB VM.

This thrashing issue arises because TPP uses NUMA balancing hints [29] to choose promotion candidates and is not aware of the global memory access distribution. We repeat the single-application performance study in §3.2 with TPP. The results show that 17% of the workloads experience more than 50% slowdown with TPP because of thrashing (Figure 11). Although software tiering can measure the global access distribution and only migrate pages when the distribution is skewed to avoid thrashing, the CPU overhead of such global telemetry is prohibitive without guest cooperation, due to the lack of spatial locality in the guest physical memory address space (§2.3). In addition, even with global telemetry, the larger page sizes used with virtualization still make software tiering less effective, as they may average out the skewness in memory access distribution (§2.3).

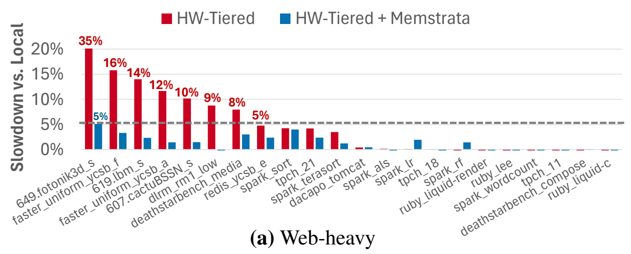
In summary, the comparison with TPP matches the results of recent work [76] indicating that hardware-based memory tiering's low overhead and cacheline-level granularity typically provide superior performance to software-based tiering. Therefore, in the rest of our experiments we focus on comparing Intel® Flat Memory Mode with and without Memstrata.

For all six experiments in Figure 10 Memstrata is able to significantly reduce the slowdown experienced by the outlier application to near 5% or less, with minimal impact to the other non-memory-sensitive applications. The max CPU overhead of Memstrata across all workload combinations is 4% of a single core, including running the ML model. The max memory overhead of Memstrata is 110 MB. The results show that Memstrata can accurately identify the outlier VM and migrate dedicated local memory pages to reduce its slowdown, without affecting the performance of other VMs.

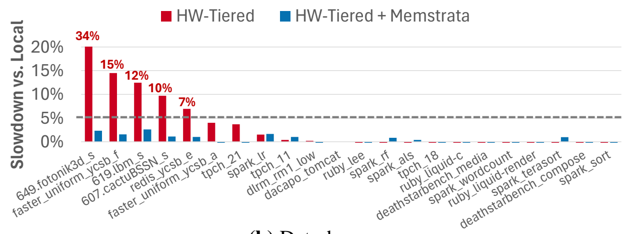
Higher outlier ratio. To understand the limits of Memstrata, we consider a server that hosts a disproportionate ratio of outlier workloads. We consider two combinations: one with two outliers (Figure 12a), and another with four outliers (Figure 12b). In both experiments, Memstrata significantly improves outlier performance. However, is not able to reduce the slowdown for all outliers to below 5% when four outliers exist. This is because 33% of dedicated local DRAM (i.e., 26.4 GB) is insufficient to accommodate the memory needs of four outlier VMs (56 GB). We verify this by repeating the experiment with 67% dedicated local DRAM. In this configuration Memstrata removes all the outliers (Figure 12c).

Dynamic VM arrivals. We evaluate Memstrata in a more complex setting where VMs are continuously launched and terminated. We again use the three workload combinations described above. Whenever a workload of one type finishes, we start a new VM with a workload selected from the same type. The experiment keeps running until all the workloads have been run at least once. We measure the application performance with and without Memstrata. For the workloads that have finished multiple times, we report its average performance across all the completed runs.

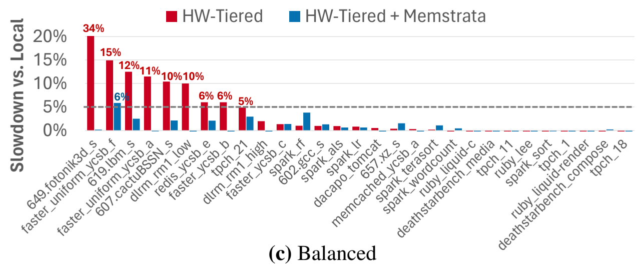
Figure 13 shows that Memstrata can significantly reduce the slowdown of the outliers in such dynamic environments under all three workload mixes. The results demonstrate that Memstrata's online outlier detection can identify the outliers on-the-fly and dynamically migrate dedicated local memory pages to reduce their slowdown. We conclude that a combi-



(a) Web-heavy



(b) Data-heavy



(c) Balanced

Figure 13: Application slowdown in realistic environments with three different workload mixes.

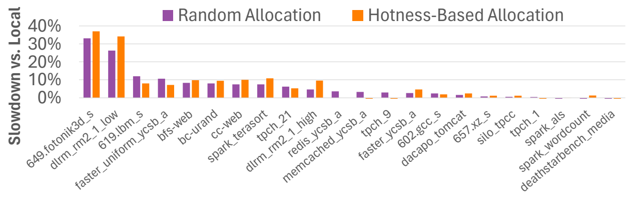
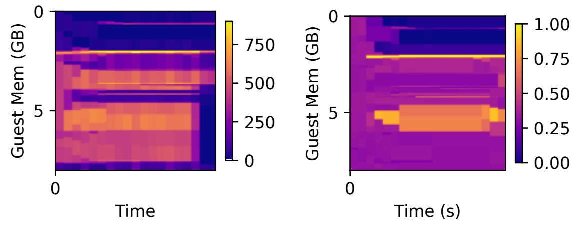


Figure 14: Slowdown of random and hotness-based page allocation.



(a) Memory accesses **(b)** Dedicated pages

Figure 15: Distributions of (a) memory accesses and (b) dedicated pages of `619.lbm_s` in the guest physical address space with hotness-based dedicated page allocation. In (a), the color represents #accesses captured by DAMON; in (b), the density of dedicated pages.

nation of Intel® Flat Memory Mode and Memstrata enables server memory capacity to be expanded by 1.5× using CXL at a minimal performance impact to applications.

6.2 Dedicated Memory Page Allocation

To understand how the allocation of dedicated local memory pages within a VM affects performance, we compare random page allocation (the default) with a hotness-based approach. We study a representative set of 22 workloads from different categories. With random allocation, the 33% of dedicated pages are randomly allocated to the VM when it is launched. With the hotness-based approach, the dedicated pages are also allocated randomly at launch, but Memstrata's dynamic allocator migrates them to hot guest physical regions based on the information provided by DAMON [11]. To measure the best-case performance of the hotness-based approach, we preserve the guest memory's spatial locality by always starting in a fresh VM.

Figure 14 shows the slowdown of both random and hotness-based page allocation. The hotness-based approach provides only marginal benefits and even causes worse slowdowns for some workloads because of its overhead. Figure 15 presents the hotness information recorded by DAMON and how the page allocator moves dedicated pages within the guest physical address space for `619.lbm_s`. Although it migrates dedicated pages to the hot regions identified by DAMON, the improvement is still limited. This is because to track memory accesses, DAMON must clear PTE access bits, resulting in expensive TLB shootdowns that offset its benefits.

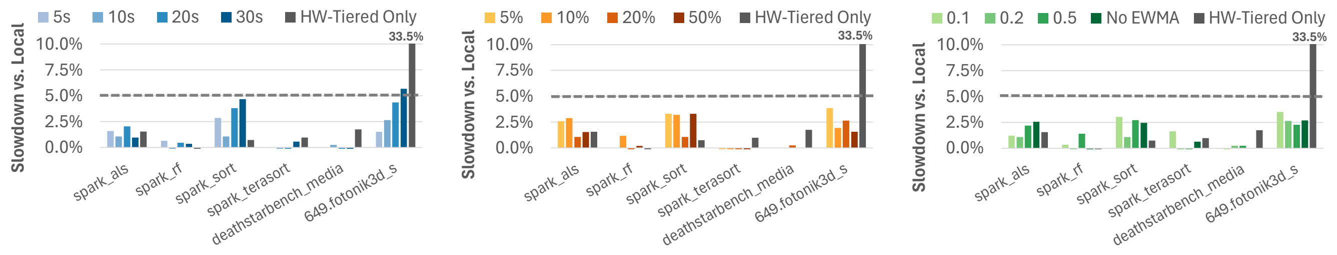
6.3 Sensitivity Analyses

Memstrata has three parameters: `timeInterval` and `stepRatio`, which control the aggressiveness of page migration, and the EWMA constant, which smoothes short-term variations of performance metrics. Figure 16 plots Memstrata's sensitivity to the three parameters using the same workloads as Figure 10e. With a lower `timeInterval` or a higher `stepRatio`, Memstrata migrates dedicated local memory pages to the outlier VM more quickly and achieves lower slowdowns (Figure 16a and Figure 16b). The default parameters (i.e., 10 s `timeInterval` and 10% `stepRatio`) have performance similar to the optimal one in this experiment, but are less aggressive and can avoid large performance fluctuations. Memstrata is not sensitive to the EWMA constant (Figure 16c).

7 Discussion

Non-virtualized environments. Most of Memstrata's components can be readily applied to non-virtualized environments. For example, per-process performance event tracking is already supported by Linux, and page migration mechanisms for both VMs and normal processes are also supported.

However, the page coloring implementation needs to be modified for non-virtualized settings. Unlike VMs, whose memory sizes do not typically change during their lifetimes, processes commonly have dynamic memory footprints. Memstrata statically allocates a fixed number of colors during VM creation, which is insufficient for processes with dynamic memory demands. Therefore, we need to augment the page coloring mechanism to support on-demand color allocation. In addition, as a process continuously allocates and frees memory, allocated colors may have numerous unused pages (e.g.,



(a) `timeInterval` (s) **(b)** `stepRatio` (%) **(c)** EWMA constant (α)

Figure 16: Sensitivity analyses of Memstrata under a data-heavy workload combination.

if a process allocates a large amount of memory but then frees half of it later). Such fragmentation can lead to insufficient free colors for new memory allocations. We therefore need to implement a compaction mechanism to reclaim colors from processes exhibiting significant fragmentation.

Intel® Flat Memory Mode with other memory ratios. In principle, Intel® Flat Memory Mode could support other ratios between local memory and CXL memory such as 1:2 or 1:4. We are not prepared to discuss whether such ratios will be available in a future version. There are also associated costs such as requiring more tag bits for bookkeeping.

Detecting outliers at the VM scheduler. Similar to Pond [78], the VM scheduler can correlate historical performance event measurements with a new VM allocation request by matching the customer ID, VM type, and location. Based on the historical information, the scheduler can perform an initial outlier detection to decide the memory type for the new VM. Additionally, the VM scheduler can also run online outlier detection after the VM is allocated and can live migrate the VM if the initial outlier detection proves to be inaccurate.

Adapting to other slowdown thresholds. Adapting to slowdown thresholds other than 5% requires retraining the random forest model using the performance events and the corresponding slowdowns of various workloads. Since the random forest model is lightweight and does not require any GPU, retraining the model incurs only minimal overhead and can be completed within a few seconds.

8 Related Work

Most prior work relies on software to place data across memory tiers [45, 70, 99, 100, 100], whereas Memstrata combines hardware tiering with a lightweight software layer. HeMem [89] and MEMTIS [74] are recent systems that use Intel PEBS to track memory accesses. Unfortunately, PEBS is not compatible with virtualized environments. In addition, unlike MEMTIS, which balances the TLB benefits of huge pages with the granularity of data placement by dynamically splitting huge pages, Memstrata achieves both low TLB cost and fine-grained data placement without sacrificing either. TPP [84] relies on LRU and NUMA balancing hints [29] to track memory accesses, but incurs high slowdowns (§6.1) and significant CPU overhead (§2.3).

Three prior systems explore software-managed tiered memory in multi-tenant environments. Unfortunately, they are not available for comparison. TMTS [61] is a memory tiering system deployed in Google's datacenters. We believe TMTS is overly conservative and requires large amounts of local memory. Pond [78] statically places VMs into a CXL-based memory pool based on predictions of slowdowns. Pond uses VM live migration to mitigate outliers, which impacts VM performance and thus must be applied conservatively. vTMM [90] is a dynamic software tiering memory management system for VMs. We believe vTMM suffers from overheads similar to other software-based systems (§2.3). Memstrata differs from all three systems due to its unique combination of hardware and software tiering in the same system.

2LM is a hardware-managed tiered memory system for Intel® Optane™ NVM, using DRAM as an inclusive direct-mapped cache of NVM. In contrast, Intel® Flat Memory Mode uses exclusive caching, and 2LM lacks the cross-VM isolation provided by Memstrata. Other hardware approaches have been proposed for DRAM and high-bandwidth memory [68, 75, 85, 103]; some Intel processors support a hardware-managed "cache mode" that uses HBM as a cache for DRAM [55, 59].

9 Conclusions

We presented a new hardware-based CXL tiering system, Intel® Flat Memory Mode, combined with a software stack, Memstrata. The combination provides performance similar to local DRAM across a wide range of workloads. Consequently, they enable expanding server memory capacity by 1.5× with minimal impact to performance. We believe there remain many open research challenges in deploying CXL in virtualized environments, including fairness in inter-VM resource allocation policies, guest cooperation for tiered memory, and using device-side hotness tracking to reduce page conflicts.

Acknowledgments

We would like to thank our shepherd, Marcos K. Aguilera, and the anonymous reviewers for their helpful comments. We also thank Phoebe Lu and Helen Chu for their contributions at the early stage of this work. This work was supported by an Intel gift and NSF grants CNS-1845853, CNS-2104243, CNS-2104292, and CNS-2143868.

References

[1] 2MB Large Memory Pages for Hypervisor and Guest Operating System. https://docs.vmware.com/en/VMware-Cloud-on-AWS/services/vmc-aws-performance/GUID-5658E64F-5606-4363-A18D-9E9175D107F0.html.

[2] AMD Bergamo and Genoa-X EPYC server CPUs crush the competition with sheer performance and efficiency dominance. https://wccftech.com/amd-bergamo-genoa-x-epyc-server-cpus-crush-competition-sheer-performance-efficiency-dominance/. [Accessed 11/14/2023].

[3] AMD, Meta are working on revolutionary tech that could recycle petabytes worth of RAM. https://www.techradar.com/pro/amd-meta-are-working-on-revolutionary-tech-that-could-recycle-petabytes-worth-of-ram-cxl-could-help-save-hyperscalers-tens-of-millions-of-dollars-while-improving-performance. [Accessed 11/14/2023].

[4] AMD Research Instruction Based Sampling Toolkit. https://github.com/jlgreathouse/AMD_IBS_Toolkit.

[5] ARM - Performance Monitor Unit. https://developer.arm.com/documentation/ddi0500/d/performance-monitor-unit.

[6] AWS EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/.

[7] Azure / Virtual Machines / General purpose virtual machine sizes. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-general.

[8] Compute Express Link (CXL). https://www.computeexpresslink.org/.

[9] CXL Members. https://www.computeexpresslink.org/members.

[10] CXL Smart Memory Controllers. https://www.asteralabs.com/products/cxl-memory-platform/.

[11] DAMON: Data Access MONitor. https://www.kernel.org/doc/html/v5.19/vm/damon/index.html.

[12] Google Cloud Platform / General-purpose machine family for Compute Engine. https://cloud.google.com/compute/docs/general-purpose-machines.

[13] HiBench Suite. https://github.com/Intel-bigdata/HiBench.

[14] Intel Announces 288-Core Sierra Forest CPU, 5th-Gen Xeon. https://www.tomshardware.com/news/intel-announces-288-core-processor-5th-gen-xeon-arrives-december-14.

[15] Intel FPGA Compute Express Link (CXL) IP. https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interface-protocols/cxl-ip.html.

[16] Intel Performance Counter Monitor. https://www.intel.com/content/www/us/en/developer/articles/tool/performance-counter-monitor.html.

[17] Intel® Optane™ Persistent Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[18] Intel® Optane™ Persistent Memory Start Up Guide. https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel_Optane_Persistent_Memory_Start_Up_Guide.pdf.

[19] Intel® Memory Latency Checker (Intel® MLC). https://www.intel.com/content/www/us/en/download/736633/intel-memory-latency-checker-intel-mlc.html.

[20] Introduction to Cache Allocation Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html.

[21] Kernel Self Protection Project/Recommended Settings. https://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project/Recommended_Settings.

[22] Leo CXL Smart Memory Controllers. https://www.asteralabs.com/products/leo/leo-cxl-memory-connectivity-controllers/.

[23] Memcached - A distributed memory object caching system. http://memcached.org.

[24] Memory management notifiers. https://lwn.net/Articles/266320/.

[25] Meta AMD CXL 2.0 Memory Expansion Demo at OCP Summit 2023. https://www.servethehome.com/meta-amd-cxl-2-0-memory-expansion-demo-at-ocp-summit-2023/.

[26] Microchip introduces new CXL smart memory controllers for data center computing enabling modern CPUs to optimize application workloads. https://www.microchip.com/en-us/about/news-releases/products/cxl-smart-memory-controllers. [Accessed 11/14/2023].

[27] Micron CZ120 memory expansion module. https://www.micron.com/solutions/server/cxl.

[28] Montage Technology Delivers the World's First CXL Memory eXpander Controller. https://www.montage-tech.com/Press_Releases/20220506.

[29] NUMA balancing: optimize memory placement for memory tiering system. https://lwn.net/Articles/849095/.

[30] [PATCH 0/5] Transparent Page Placement for Tiered-Memory. https://lore.kernel.org/all/cover.1637778851.git.hasanalmaruf@fb.com/.

[31] [PATCH] AMD perf PMU events for AMD Family 17h. https://lore.kernel.org/lkml/3ee15066-429e-b0f2-1255-aab100fad472@suse.cz/.

[32] Pathfinding Cloud Architecture for CXL with Dan Ernst of Microsoft Azure. https://gestaltit.com/all/stephen/pathfinding-cloud-architecture-for-cxl-with-dan-ernst-of-microsoft-azure-utilizing-tech-4x10/.

[33] PostgreSQL Database Management System. https://www.postgresql.org/.

[34] [QEMU-devel] [PATCH] Call MADV_-HUGEPAGE for guest RAM allocations. https://lists.nongnu.org/archive/html/qemu-devel/2012-10/msg01012.html.

[35] Recording Hardware Performance (PMU) Events. https://learn.microsoft.com/en-us/windows-hardware/test/wpt/recording-pmu-events.

[36] Redis, an in-memory data structure store. http://redis.io.

[37] Samsung Electronics Introduces Industry's First 512GB CXL Memory Module. https://semiconductor.samsung.com/news-events/news/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module/.

[38] Samsung's memory-semantic CXL SSD brings a 20x performance uplift. https://www.tomshardware.com/news/samsung-memory-semantic-cxl-ssd-brings-20x-performance-uplift. [Accessed 11/14/2023].

[39] Shopify/yjit-bench: Set of benchmarks for the YJIT CRuby JIT compiler and other Ruby implementations. https://github.com/Shopify/yjit-bench.

[40] SPEC CPU 2017. https://www.spec.org/cpu2017/.

[41] TPC Benchmark C (TPC-C). http://www.tpc.org/tpcc/.

[42] TPC Benchmark H (TPC-H). https://www.tpc.org/tpch/.

[43] VMware vCloud NFV OpenStack Edition / Tuning vCloud NFV for Data Plane Intensive Workloads / Huge Pages. https://docs.vmware.com/en/VMware-vCloud-NFV-OpenStack-Edition/3.0/vmwa-vcloud-nfv30-performance-tunning/GUID-1F05987F-012B-4BC4-9015-CDE3C991C68C.html.

[44] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.

[45] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 631–644, New York, NY, USA, 2017. Association for Computing Machinery.

[46] Emmanuel Amaro, Stephanie Wang, Aurojit Panda, and Marcos K Aguilera. Logical memory pools: Flexible and local disaggregated memory. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 25–32, 2023.

[47] Hugo Barbalho, Patricia Kovaleski, Beibin Li, Luke Marshall, Marco Molinaro, Abhisek Pan, Eli Cortez, Matheus Leao, Harsh Patwari, Zuzu Tang, et al. Virtual machine allocation with lifetime predictions. *Proceedings of Machine Learning and Systems*, 5, 2023.

[48] Scott Beamer, Krste Asanović, and David Patterson. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[49] Daniel S. Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D. Hill, and Ricardo Bianchini. Design tradeoffs in CXL-based memory pools for public cloud platforms. *IEEE Micro*, 43(2):30–38, 2023.

[50] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. 42(2):26–35, mar 2008.

[51] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, 2006.

[52] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[53] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, 2021.

[54] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 275–290, New York, NY, USA, 2018. Association for Computing Machinery.

[55] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. BATMAN: Techniques for maximizing system bandwidth of memory systems with stacked-DRAM. In *Proceedings of the International Symposium on Memory Systems*, pages 268–280, 2017.

[56] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, March 2016. USENIX Association.

[57] Ampere Computing. AmpereOne 192-core CPU family product brief. https://amperecomputing.com/briefs/ampereone-family-product-brief. [Accessed 11/14/2023].

[58] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[59] Daniel DeLayo, Kenny Zhang, Kunal Agrawal, Michael A Bender, Jonathan W Berry, Rathish Das, Benjamin Moseley, and Cynthia A Phillips. Automatic HBM management: Models and algorithms. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 147–159, 2022.

[60] ONNX Runtime developers. ONNX runtime. https://onnxruntime.ai/, 2021. Version: x.y.z.

[61] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery.

[62] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[63] Udit Gupta, Mariam Elgamal, Gage Hills, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. ACT: Designing sustainable computer systems with an architectural carbon modeling tool. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 784–799, 2022.

[64] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM Allocation Service at Scale. In *USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[65] Mark Hildebrand, Julian T. Angeles, Jason Lowe-Power, and Venkatesh Akella. A case against hardware managed DRAM caches for NVRAM based systems. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–204, 2021.

[66] Intel. 4th gen Intel Xeon scalable processors. https://download.intel.com/newsroom/2023/data-center-hpc/4th-Gen-Xeon-Scalable-Product-Brief.pdf, 2023.

[67] Rishabh Jain, Scott Cheng, Vishwas Kalagi, Vrushabh Sanghavi, Samvit Kaul, Meena Arunachalam, Kiwan

Maeng, Adwait Jog, Anand Sivasubramaniam, Mahmut Taylan Kandemir, and Chita R. Das. Optimizing cpu performance for recommendation systems at-scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[68] Djordje Jevdjic, Gabriel H Loh, Cansu Kaynak, and Babak Falsafi. Unison cache: A scalable and effective die-stacked DRAM cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–37. IEEE, 2014.

[69] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 10(4):338–359, nov 1992.

[70] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for multi-tiered memory systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 715–728. USENIX Association, July 2021.

[71] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[72] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[73] Seok-Hee Lee. Technology scaling challenges and opportunities of memory devices. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 1–1. IEEE, 2016.

[74] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. MEMTIS: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 17–34, 2023.

[75] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W Lee. A fully associative, tagless DRAM cache. *ACM SIGARCH computer architecture news*, 43(3S):211–222, 2015.

[76] Baptiste Lepers and Willy Zwaenepoel. Johnny cache: the end of DRAM cache conflicts (in tiered main memory systems). In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 519–534, Boston, MA, July 2023. USENIX Association.

[77] Philip Levis, Kun Lin, and Amy Tai. A case against CXL memory pooling. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 18–24, 2023.

[78] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[79] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan RK Ports, Irene Zhang, Ricardo Bianchini, Haryadi S Gunawi, and Anirudh Badam. LeapIO: Efficient and portable virtual NVMe storage on arm SOCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 591–605, 2020.

[80] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and Ponnuswamy Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378. IEEE, 2008.

[81] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.

[82] Sihang Liu, Suraaj Kanniwadi, Martin Schwarzl, Andreas Kogler, Daniel Gruss, and Samira Khan. Side-channel attacks on Optane persistent memory. In *32nd USENIX Security Symposium 2023*. USENIX Association, 2023.

[83] Jialun Lyu, Jaylen Wang, Kali Frost, Chaojie Zhang, Celine Irvene, Esha Choukse, Rodrigo Fonseca, Ricardo Bianchini, Fiodar Kazhamiaka, and Daniel S. Berger. Myths and Misconceptions Around Reducing Carbon Embedded in Cloud Platforms. In *HotCarbon Workshop*, 2023.

[84] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[85] Sparsh Mittal and Jeffrey S Vetter. A survey of techniques for architecting DRAM caches. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1852–1863, 2015.

[86] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[87] Dave Patterson. 10 lessons from a decade of TPUs and ML's carbon footprint. https://www.youtube.com/watch?v=--z1cmq1BCw, 2023.

[88] Aleksandar Prokopec, Andrea Rosa, David Leopoldseder, Gilles Duboscq, Petr Tuuma, Martin Studener, Lubomir Bulej, Yudi Zheng, Alex Villazon, Doug Simon, et al. Renaissance: A modern benchmark suite for parallel applications on the JVM. In *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 11–12, 2019.

[89] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable tiered memory management for big data applications and real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[90] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. vTMM: Tiered memory management for virtual machines. EuroSys '23, page 283–297, New York, NY, USA, 2023. Association for Computing Machinery.

[91] Debendra Das Sharma, Robert Blankenship, and Daniel S. Berger. An introduction to the Compute Express Link (CXL) interconnect. *ACM Computing Surveys (CSUR)*, 2024.

[92] Shigeru Shiratake. Scaling and performance challenges of future DRAM. In *2020 IEEE international memory workshop (IMW)*, pages 1–3. IEEE, 2020.

[93] Livio Soares, David Tam, and Michael Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 258–269. IEEE, 2008.

[94] Lisa Su. Amd unveils workload-tailored innovations and products at the accelerated data center premiere. https://www.amd.com/en/press-releases/2021-11-08-amd-unveils-workload-tailored-innovations-and-products-the-accelerated, November 2021.

[95] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing shared L2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 26–33, 2007.

[96] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[97] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the tenth european conference on computer systems*, pages 1–17, 2015.

[98] Jaylen Wang, Daniel S. Berger, Fiodar Kazhamiaka, Celine Irvene, Chaojie Zhang, Esha Choukse, Kali Frost, Rodrigo Fonseca, Brijesh Warrier, Chetan Bansal, Jonathan Stern, Ricardo Bianchini, and Akshitha Sriraman. Designing cloud servers for lower carbon. In *ISCA*, June 2024.

[99] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 609–621, New York, NY, USA, 2022. Association for Computing Machinery.

[100] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 331–345, New York, NY, USA, 2019. Association for Computing Machinery.

[101] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44. IEEE, 2014.

[102] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: A dynamic cache partitioning system using page coloring. PACT '14, page 381–392, New York, NY, USA, 2014. Association for Computing Machinery.

[103] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, Onur Mutlu, and Srinivas Devadas. Banshee: Bandwidth-efficient DRAM caching via software/hardware cooperation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–14, 2017.

[104] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for inmemory cluster computing. In *USENIX NSDI*, 2012.

[105] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102, 2009.