HeteroOS - OS Design for Heterogeneous Memory Management in Datacenter

Sudarsun Kannan[1] Ada Gavrilovska[2] Vishal Gupta[3] Karsten Schwan[2]
[1]Department of Computer Sciences, University of Wisconsin-Madison
[2]School of Computer Science, Georgia Tech,
[3]VMWare
{sudarsun@cs.wisc.edu},{ada@cc.gatech.edu},{vishalg@vmware.com}

ABSTRACT

Heterogeneous memory management combined with server virtualization in datacenters is expected to increase the software and OS management complexity. State-of-the-art solutions rely exclusively on the hypervisor (VMM) for expensive page hotness tracking and migrations, limiting the benefits from heterogeneity. To address this, we design **HeteroOS**, a novel application-transparent OS-level solution for managing memory heterogeneity in virtualized system. The HeteroOS design first makes the guest-OSes heterogeneity-aware and then extracts rich OS-level information about applications' memory usage to place data in the 'right' memory avoiding page migrations. When such pro-active placements are not possible, HeteroOS combines the power of the guest-OSes' information about applications with the VMM's hardware control to track for hotness and migrate only performance-critical pages. Finally, HeteroOS also designs an efficient heterogeneous memory sharing across multiple guest-VMs. Evaluation of HeteroOS with memory, storage, and network-intensive datacenter applications shows up to 2x performance improvement compared to the state-of-the-art VMM-exclusive approach.

CCS CONCEPTS

• **Software and its engineering** → *Virtual machines*; *Virtual memory*; *Main memory*; • **Computer systems organization** → Processors and memory architectures; • **Hardware** → Non-volatile memory;

KEYWORDS

Heterogeneous Memory, Operating Systems, Virtual Memory, Virtualization, Hypervisor, 3D-stacked DRAM, Non-volatile memory

ACM Reference format:
Sudarsun Kannan[1] Ada Gavrilovska[2] Vishal Gupta[3] Karsten Schwan[2]
[1]Department of Computer Sciences, University of Wisconsin-Madison [2]School of Computer Science, Georgia Tech, [3]VMWare . 2017. HeteroOS - OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 14 pages.
https://doi.org/10.1145/3079856.3080245

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISCA '17, June 24-28, 2017, Toronto, ON, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4892-8/17/06...$15.00
https://doi.org/10.1145/3079856.3080245

Property	Stacked-3D	DRAM	NVM (PCM)
Density	1x	4x-16x	16x-64x
Load latency (ns)	30-50	60	150
Store latency (ns)	30-50	60	300-600
BW (GB/sec)	120-200	15-25	2

Table 1: Heterogeneous memory characteristics [7, 15, 50].

1 INTRODUCTION

To address the DRAM capacity scalability bottlenecks [42, 63] and the need for lower access latency and higher bandwidth, researchers and commercial vendors are exploring alternative memory technologies such as 3D-stacked DRAM and non-volatile memory (NVM). As shown in Table 1, the technologies differ significantly in latency, bandwidth, endurance. For instance, as shown in Table 1, byte-addressable non-volatile memories (NVMs) such as phase change memory (PCM), are expected to offer higher capacity than DRAM, but with higher read (2x) and write (up to 5x) latency, and lower bandwidth (5x-10x) [15, 36, 49]. Conversely, on-chip stacked 3D-DRAM [6, 7, 21, 29, 41] is expected to increase memory bandwidth by 8x-14x, but with a 2x-4x lower capacity than DRAM. These differences indicate that a single memory technology will not solve all memory-related bottlenecks for an application and future systems will embrace memory heterogeneity [4, 50], thus increasing the software-level management complexity. In this paper, we study the software-level challenges in supporting memory heterogeneity for datacenter systems.

At a high-level, structuring heterogeneous memories as OS-level NUMA nodes is a natural fit and provides an opportunity to reuse existing OS and application-level abstractions [15, 43]. However, several fundamental differences exist in the homogeneous NUMA and heterogeneous memory systems. First, heterogeneous memory technologies have significantly different latency and bandwidth, unlike DRAM-based NUMA. Second, for homogeneous NUMA, the OS-level management aims to increase data locality by increasing CPU access to the data in the local memory socket. In contrast, for heterogeneous memory, the challenge is (a) to identify performance-critical data and place them in the fastest memory, and (b) to maximize the utilization of the fast memory with limited capacity [45]. For heterogeneous memory systems, the performance impact due

to incorrect memory placement of critical data can be substantial compared to the traditional NUMA systems with less than 50% overheads.

Prior heterogeneous memory research either relies on significant hardware changes to the memory subsystem such as the memory controller, TLB, and cache [7, 43] or to the applications [3, 15, 32, 37]. While the hardware changes can be time-consuming, the application-level changes require developers to add explicit memory placement or migration logic. Furthermore, application-level techniques also lack a holistic view of the system in multi-tenant or virtualized datacenter systems. Even today, most guest-OSes in virtualized environments lack fine-grained NUMA topology awareness or data placement control, requiring methods like HeteroVisor [24] to hide heterogeneity from the guest-OSes and manage heterogeneity in the hypervisor (VMM). Such approaches solely rely on the hypervisor (VMM) for expensive hotness tracking of the VMs' memory and sub-optimal page migration (moving hot pages to the faster memory). All of the above approaches fail to exploit the rich guest-OS-level information about the applications and their memory use. To address these problems, we design and implement **HeteroOS** – a performance-efficient OS design for an application-transparent heterogeneous memory management in virtualized systems. The key design ideas and contributions are as follows.

Guest-OS heterogeneity awareness and memory placement. First, in HeteroOS, we make the guest-OSes memory heterogeneity-aware and then extract the rich information from guest-OS about how applications use different memory pages (e.g., heap, IO buffer caches). We combine this information with guest-OS heterogeneity awareness to provide an application-transparent OS-level memory placement avoiding expensive page migrations.

Against the conventional OS memory management methods that always prioritize heap to the faster memory (e.g., DRAM) [9] when its capacity is constrained, we show that for heterogeneous memory systems, it is critical to equally prioritize heap and I/O pages to the faster memory for accelerating in-memory, storage, and network-intensive applications. We also design *HeteroOS-LRU*, a fast memory contention resolution method.

Coordination with VMM. Guests-OSes in virtualized systems lack a holistic view of other guest-VMs and direct hardware control required for privileged operations such as hot page tracking. We address this by designing a VMM-guest coordinated management where the VMM performs hotness tracking, and the guest-OSes guide the VMM with their deeper view of application-specific information.

Heterogeneous memory sharing and fairness. Efficient and fair sharing of heterogeneous memory across multiple guest-VMs is important. Hence, we design a novel multi-resource sharing by extending Dominant Resource Fairness (DRF) [19]. DRF provides Strategy-proofness and Pareto efficiency. We also extend the traditional memory ballooning with heterogeneous memory support [62] for enabling memory overcommit.

Extensive evaluation. We evaluate HeteroOS with a wide range of memory, storage, and network-intensive datacenter applications. The guest-OS level management combined with VMM-guest coordinated approach shows up to 3x improvement in performance compared to always using slow memory, and up to 2x compared to the state-of-the-art VMM-exclusive management.

Application	Description	Perf. metric
GraphChi [34]	Pagerank using Orkut social graph, 8 million nodes, 500 million edges [16]	time(sec)
X-Stream [56]	Edge-centric graph processing and uses same input as GraphChi	time(sec)
Metis [5]	Shared memory mapreduce that optimizes Phoenix [53], 4GB crime dataset, 8 mapper-reducer threads	time(sec)
LevelDB [18]	Google's DB for bigtable, SQlite bench with 1M keys	throughput (MB/s)
Redis [57]	Popular key-value store with support for persistence, analyzed with Redis benchmark 4 millions ops., 80% get	requests per sec
NGinx [28]	State of the art webserver, 1 million static, dynamic, images webpages	requests per sec

Table 2: Datacenter applications.

Factor	L:1, B:1	L:2, B:2	L:5, B:5	L:5, B:12
Latency (ns)	60	128	354	960
BW (GB/s)	24	12.4	5.1	1.38

Table 3: L:x, B:y indicates the latency increase factor x, and bandwidth reduction factor y respectively.

In the remainder of this paper, in Section 2, we first briefly provide a background on heterogeneous memory technologies and our emulation method. In Section 2, we provide an empirical evidence motivating the need for an OS-level design for memory heterogeneity management. In Section 3, we discuss the principles of HeteroOS, followed by the design and implementation of the guest-OS level management. In Section 4, we discuss the coordinated management approach, followed by the resource sharing and fairness mechanism. In Section 5, we evaluate HeteroOS, and finally, present the related work and conclusions in Section 6 and Section 7 respectively.

2 BACKGROUND AND MOTIVATION

Technologies such as PCM and spin-torque transfer (STTRAM) are expected to provide 4x-8x higher capacity and lower cost per gigabyte [14, 15] compared to DRAM. However, recent industrial projections, and prior research (see Table 1) show up to 2x higher read latency, 5x-8x higher write latency, and up to 10x lower bandwidth [15, 59, 65]. When using NVM as main memory [1, 35, 40, 49], the processor cache is expected to play a significant role in reducing the write latency cost. Next, the endurance – the lifetime of these technologies – is expected to be significantly lower compared to DRAM, which can be critical when using them as main

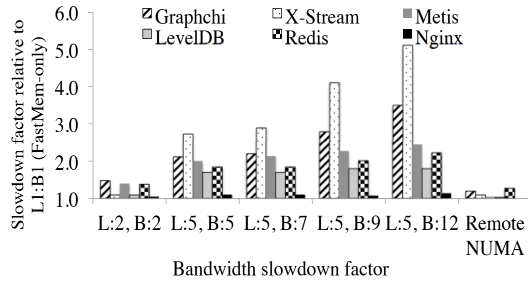

Figure 1: Bandwidth and latency sensitivity: Bars show slowdown factor relative to FastMem(L:1,B:1). Remote NUMA bars represent FastMem on a remote socket. Analysis using Intel(R) Xeon(R) CPU, 16 cores X5560 with 16MB cache.

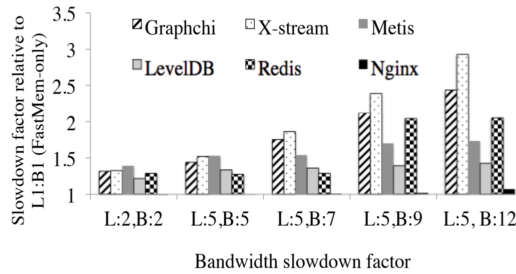

Figure 2: Intel NVM emulator bandwidth and latency sensitivity. System has Intel(R) Xeon(R) CPU, 16 cores, E5-4620 v2 with 48MB cache.

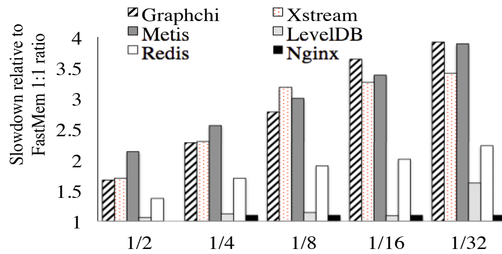

Figure 3: FastMem capacity impact.

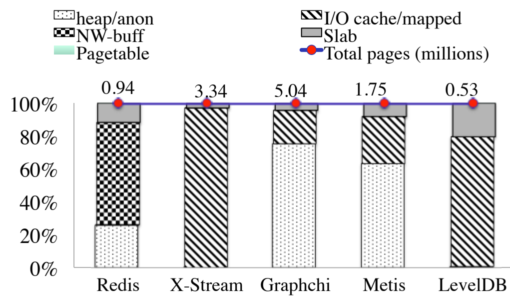

Figure 4: Application memory page distribution.

memory. Prior work has proposed wear-levelling solutions ranging from device-level FTL-like mechanisms [38, 39], or memory-controller enhancements [47, 55] such as introducing large SRAM cache [31] or DRAM write buffers. Application-level techniques aim to reduce hot page placement in the NVM [27, 61].

In contrast to NVM, on-chip memory such as stacked 3D-DRAM, Hybrid Memory Cube (HMC) and High Bandwidth Memory (HBM) [7, 43] are expected to provide 10x higher bandwidth and 1.5x lower latency, but with 2x-4x lower capacity compared to DRAM. Hence, heterogeneous memory is bound to increase the memory management complexity of the software stack.

2.1 Assumptions and emulation

Lack of commercially available heterogeneous memory technologies and a wide range of performance parameters and endurance characteristics quoted by different sources presents a methodological challenge for a meaningful study to understand the impact of heterogeneity. Using cycle-accurate instruction-level simulators [7, 13, 43] in both user and the OS stack for long-running applications is not practical. Hence, in this paper, we consider two generic types of memory (1) **FastMem** - high bandwidth and low latency, and limited capacity memory, and (2) **SlowMem** - a low bandwidth, high latency, but a large capacity memory.

Emulation. To emulate FastMem and SlowMem, we modify the PCI-based thermal registers to throttle the per-socket DRAM bandwidth and latency as listed in Table 3, where the columns (L:x, B:y) represent the factor of increase in latency, and a factor of reduction in bandwidth relative to the DRAM. In this paper, we use the DRAM (L:1, B:1) as the FastMem baseline, and reduce its bandwidth by up to 12x and increase the latency by up to 5x to emulate SlowMem. Prior studies [24, 26, 32] have also shown that throttling does not have side effects other than the bandwidth and latency impact based on the application's memory intensity. We also verify our analysis using Intel's persistent memory emulator [14] that emulates NVM's read-write latency using a microcode patch and bandwidth emulation via throttling. Due to restricted (user-level) access to this emulator, we only use this platform for analysis. We consider generic FastMem and SlowMem in our study, but even for technologies such as NVM, when used as a heap for non-persistent writes, store operations are posted to write-back cacheable memory. Hence, the size of the cache, NVM bandwidth, and both read-write latency can become critical as shown by Dulloor et al. [15].

2.2 Applications and analysis

To understand the implications of memory heterogeneity on applications, we next analyze several real-world datacenter applications listed in Table 2 by emulating heterogeneity. Unlike prior research that mainly targets in-memory applications; we study applications with high variability in their memory, storage, and network. This includes graph analytics, in-memory data stores, map-reduce computations, popular databases, and a web server [16].

Memory latency and bandwidth sensitivity. Figure 1 shows the sensitivity of applications when the SlowMem bandwidth and latency factors (L:x, B:y) are varied in the x-axis. The y-axis shows the slowdown factor of applications compared to when running exclusively in FastMem (L:1, B:1). Table 4 shows the memory intensity of the applications in MPKI (misses per kilo instructions).

App.	Graphchi	X-Stream	Metis	LevelDB	Redis	Nginx
MPKI	27.4	24.8	14.9	4.7	11.1	2.1

Table 4: Memory intensity of applications in MPKI.

Observation 1. Applications show higher sensitivity to latency variations compared to bandwidth.

Memory-intensive applications such as Graphchi, X-Stream, and Metis (see Table 4) show higher sensitivity (slowdown) towards latency increase and bandwidth reduction. In contrast, the storage-intensive LevelDB and network-intensive Redis with relatively smaller working set size show lower impact. Nginx, a popular web-server, is both storage and network-intensive with a less than 60MB active working set. Hence, even exclusively placing it in a 9x SlowMem has less than 10% impact. Next, except Graphchi and X-Stream, other applications have a lower impact from bandwidth reduction compared to latency increase. This is because multi-threaded graph-compute applications process and move data in batches, thereby increasing memory traffic. However, other applications lack parallelism and memory intensity to saturate the bandwidth. For instance, when the bandwidth is reduced from (L:5, B:5) to (L:5, B:12) with latency as a constant, Metis, Redis and LevelDB show a relatively lower slowdown. Hence, timely allocation of FastMem pages even without page migrations can be critical.

In Figure 2, we analyze the same set of applications on Intel's NVM emulator that varies the write latency on a cache miss and the bandwidth. The bandwidth and latency sensitivity trends of the applications match the analysis with our emulation mechanism. The Intel emulator platform has a 3x larger LLC (48 MB) compared to our emulator (16MB) with newer generation (Intel IvyBridge) processors. As a result, the application slowdown factor is lower for the same workloads. Increasing the workload size showed higher application slowdown.

Observation 2. Incorrect memory placement cost in heterogeneous memory is significant compared to traditional NUMA systems.

In Figure 1, we show the impact of incorrect memory placement in a NUMA system by making applications access a remote NUMA socket FastMem for all the access. We observe that even for the most memory-intensive (Graphchi, X-stream) or in-memory applications (Redis), the slowdown is less than 30% compared to significantly higher slowdowns in heterogeneous memory.

Impact of FastMem capacity. Figure 3 shows the sensitivity towards FastMem capacity for an L:5, B:9 configuration. The x-axis represents the ratio of FastMem to SlowMem capacity. For 1/2 ratio, we use 4GB of FastMem and 8GB of SlowMem. Figure 4 shows the memory page distribution and the total memory pages used for the same set of applications.

Observation 3. On-demand page allocation is not only useful for the heap but also for other OS subsystems.

Providing support for direct on-demand memory allocation to a faster memory can significantly benefit large scale capacity-intensive applications as they frequently allocate and release memory. For example, as shown in Figure 3, the capacity-intensive Graphchi and X-Stream suffer less than 2x slowdown even with a 1/2 FastMem-SlowMem ratio. But importantly, apart from the heap use, I/O- (storage and network) intensive applications frequently allocate and release memory via their OS subsystem for the filesystem page cache, network, and storage kernel buffers (slab cache). On-demand allocation of these pages to FastMem can significantly improve application performance. For example, as shown in Figure 4, a significant number of OS pages are allocated for the filesystem-intensive LevelDB, the page-cache intensive X-Stream (as it maps input graph to page cache), and the network kernel buffer-intensive Redis. Note that most OS pages are short-lived and have high reuse, as they are released once an I/O is complete. These applications show significantly lower impact even as the FastMem capacity ratio is reduced from 1/2 to 1/16. Prior studies only target heap-intensive applications and miss the opportunity for a fine-grained OS-level placement.

2.3 State-of-the-art and limitations

Recent studies on heterogeneous memory management [7, 13, 24, 36, 43, 49] have extensively used page hotness-tracking and migration. The earliest hotness-tracking mechanism was proposed by P.Denning [12] for disk swapping. More recently, Gupta et al. [24] proposed HeteroVisor, an approach for managing heterogeneous memory in virtualized systems. HeteroVisor is a guest-OS transparent and VMM-exclusive solution that completely relies on page hotness-tracking and migration without any proactive memory placement. Briefly, HeteroVisor and most software methods capture page hotness by counting the number of references to a page table entry in the hardware page table. Each entry has an 'access bit' that is set when a page is accessed. The hotness-tracking mechanism periodically scans the page table, records the value of the access bit (set to '1' if a page is referenced), and resets the bit. HeteroVisor also implements several optimizations such as batched hotness-tracking and a VMM-level page reverse map for quick page table walk, similar to non-virtualized OSes [10]. Pages that are identified as hot are promoted to FastMem, whereas least recently used hot pages are evicted to a slower memory.

Observation 4. Reactive hotness-tracking and migration approaches induce significant software overhead.

Although software-level hotness-tracking and migration provide application- and guest-OS transparency (when done in the VMM), they are expensive. First, the page table should be frequently scanned for collecting correct hotness information. Second, the hardware TLB entries should be periodically flushed even for tracking to enable a TLB translation miss and force page table reference. Third, in OSes such as Linux, during a page walk, several validity checks (discussed in Section 4) must be performed before a page is migrated. Finally, moving pages from one memory to another requires allocation of new pages, data copy, and a TLB flush, all performed by stalling a core. We evaluate these overheads in Section 5.

Observation 5. The VMM-exclusive approach lacks information and scope of an application.

The VMM's memory management data structures are coarse grained and treat the entire guest-VM as an application. Consequently, this forces the VMM to rely solely on tracking the entire guest-VM's memory and migrating hot pages, thereby adding significant software cost. In contrast, the guest-OS has rich application-specific information such as its virtual memory page usage information (e.g., anonymous, I/O, cache, and DMA), and their current state (actively used, inactive, or swapped). Guiding the VMM with the guest-OS information can reduce the monitoring scope and the associated cost.

3 HETEROOS PRINCIPLES AND DESIGN

Using the observations and the limitations of the state-of-the-art VMM-exclusive solutions, we next formulate the design principles for an efficient and application-transparent memory heterogeneity management which are both critical for faster commercial adaptation [20].

Principle 1: Providing memory heterogeneity awareness to the guest-OS is important.

Guest-OS heterogeneity awareness enables fine-grained data placement and avoids frequent migrations. We first enable the NUMA abstraction at the guest-OS, and then redesign and extend the OS-level data structures, page allocators, NUMA-related data structures and drivers for memory-type specific allocation.

Principle 2: Capturing subsystem-level page usage information enables smart memory placement.

HeteroOS exploits guest-OS heterogeneity-awareness to capture information about how memory pages are allocated and used by the application and the OS-subsystems. Using the information, HeteroOS prioritizes page placement across memories without relying on the support for VMM-level management. To reduce contention between applications and OS subsystems for FastMem pages, we design a novel guest-OS level LRU-based page replacement.

Principle 3: Supporting a coordinated guestOS-VMM management exploits the VMM's holistic view of the system.

The VMM has a holistic view and control of the system resources and the hardware. Hence, we design HeteroOS-coordinated, a coordinated management approach between the guest-OS and VMM that enables the guest-OS to delegate and guide how the VMM performs privileged operations such as hardware page table scans for tracking hotness information. The VMM also provides heterogeneous memory sharing across VMs with a novel mechanism based on Dominant Resource Fairness.

3.1 HeteroOS guest-OS design

We first discuss the design and implementation details of guest-OS virtual memory support for heterogeneity-awareness, followed by the smart memory placement mechanism and the HeteroOS's LRU page replacement. In Section 4, we discuss the VMM-guest coordinated management and the DRF-based resource sharing algorithm and implementation.

Heterogeneous memory abstraction. We design HeteroOS to provide an application-transparent heterogeneity support that leverages the existing OS abstractions and memory management functionalities without throwing away decades of research. We believe this is important for faster real-world adaptation. Therefore, we use the generic NUMA abstraction but extend it internally. First, we expose the memory types (e.g., FastMem and SlowMem) as NUMA nodes by enabling the typically disabled NUMA support for guest-VMs. We achieve this by using the Linux fake NUMA patch [54]. In Figure 5, the guest-OS layer shows a SlowMem and FastMem manager and the related heterogeneous memory components which we discuss shortly. The initial capacities of the different memory types are added to the guest's boot configuration. For distinguishing among the memory node types, a special flag is added to the node structure. Guest-VM applications also have the flexibility to map memory explicitly to the FastMem or the SlowMem with an additional *mmap()* flag, but HeteroOS is not dependent on such application-level changes.

On-demand allocation driver. In virtualized systems, during the VM boot, the boot manager initializes a guest-VM's memory and adds the VM pages under the control of the OS allocator (Buddy allocator in Linux). When the reserved pages are insufficient, the guest-OS uses an on-demand allocation balloon driver to request the VMM to increase the reservation [62]. For heterogeneous memory, HeteroOS first extends the boot allocator to initialize one NUMA node and its related data structures for each memory type. Next, for supporting overcommit and dynamically increasing the reservation of a memory type on-demand, as shown in steps 1 and 2 in Figure 5, we design a new on-demand allocation driver that provides multi-dimensional data structures required for memory-type-specific page allocation. The back-end in the VMM handles the node-specific requests and also maintains the per-node (memory type) machine page number (MFN) mapping for each of the guests. The front-end can also specify a fallback strategy when pages from a particular memory type cannot be provided. Note that in both homogeneous and heterogeneous memory systems, pages allocated by the on-demand balloon drivers are identified with a special flag, and are returned to the VMM when the system memory pressure is high.

Extending page allocators and per-CPU free list. Limiting the FastMem use only for performance-critical data is important due to its limited capacity. Hence, we make several extensions to the OS allocators. Currently, the OS statically partitions each NUMA node into three zones – a high memory zone for user-level allocation, a normal zone for kernel allocation (currently unused in 64-bit), and a DMA zone. In HeteroOS, FastMem nodes are partitioned with just one zone where both the application and OS related pages can be allocated to conserve pages. Second, FastMem pages can be allocated only using the HeteroOS page allocator to avoid other general purpose allocation by the default OS allocators (Buddy allocator in Linux). Automatic NUMA memory placement policies that are designed for increasing the CPU affinity in homogeneous systems are disabled for FastMem. Next, pages are identified with an additional 1-bit (FASTMEM, SLOWMEM) flag for specialized allocation and replacement based on the memory type. Finally, the

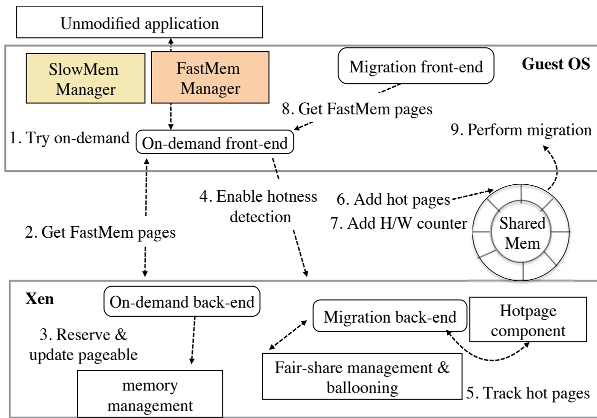

Figure 5: HeteroOS on-demand allocation, and coordinated management. In the figure, steps 1-3 show on-demand allocation, steps 4-9 show hotness-tracking and migration after the on-demand allocation fails.

Linux OS maintains a per-CPU free page list for fast allocation of pages that bypasses the fragmentation-efficient but complex Buddy allocator. However, the free CPU lists are designed for a single memory type. In HeteroOS, we redesign the per-CPU lists with a multi-dimensional (arrays of lists) support for different memory types which significantly boosts the allocation performance.

3.2 Memory placement and management

We next discuss application-transparent smart memory placement mechanism at the guest-OS level that extracts the subsystem-level page use information to prioritize FastMem allocation.

Key idea. In homogeneous memory systems, when memory is scarce, the heap page allocation requests are prioritized over allocation requests from other I/O subsystems [30] such as I/O page cache pages. Using the same model for heterogeneous memory systems with limited FastMem capacity will always force the I/O cache pages to be allocated from SlowMem resulting in a significant slowdown for I/O-intensive application. In contrast, *mapping the I/O pages also, can significantly hide the bottlenecks of slower disks and network.* Hence, HeteroOS uses demand-based prioritization across subsystems.

Demand-based FastMem prioritization. We implement a HeteroOS allocator that extends the Linux page allocator and provides OS-level heterogeneous memory allocations for the heap, I/O page cache, and the OS slab allocations. During an application execution, the allocator periodically (we use 100ms but it is configurable) extracts information such as total page allocation requests, FastMem allocation hits, and misses, for allocation requests from different subsystems for page types such as heap (anonymous), page cache, buffer cache, and kernel slab pages. When the FastMem capacity is saturated, to resolve contention across subsystems, HeteroOS employs a novel HeteroOS-LRU (discussed shortly) to evict inactive pages of any subsystem (including the heap) to prioritize allocation of page types with maximum miss ratio. Unlike the heap-only approach used in traditional homogeneous memory systems, HeteroOS's demand-based prioritization provides the following benefits.

In-memory applications (Heap-OD). For in-memory applications, most memory allocations and references (∼90%) are from the heap [15]. As a result, the demand for the heap page types is high leading to an increase in the FastMem heap miss ratio. Consequently, with our on-demand allocation, the heap page types dominate the FastMem allocation. Our results for Heap-OD show up to 180% gains compared to using only the SlowMem pages.

Storage-intensive application (Heap-IO-OD). For most storage-intensive applications such as databases and graph analytics [18, 34, 56], the I/O page cache plays a crucial role in improving the I/O throughput and application performance by leveraging the spatial and temporal locality by reading ahead I/O pages and buffering dirty blocks. During the I/O phase of an application, the page cache allocation requests are high. Using the allocation miss ratio to prioritize the page cache to FastMem can significantly improve application performance.

OS kernel buffers (Heap-IO-Slab-OD). Storage and network-intensive applications spend a significant time allocating and accessing the OS kernel buffers (slab pages). Network-intensive applications extensively use slab pages for OS-level network buffers 'skbuff' (see Redis in Figure 4), whereas storage-intensive applications allocate slab pages for the filesystem metadata which are crucial for storage performance. Prioritizing slab pages to FastMem accelerates I/O-intensive applications. Regarding prioritizing the page table pages to FastMem, as shown in Figure 4, the number of page table pages and the time spent on accessing these pages is just a small fraction of the overall application and other OS-level pages for the applications we analyzed. Placing page table pages to FastMem or SlowMem shows negligible (less than 0.5%) performance impact. We plan to explore more applications in the future.

3.3 Resolving contention with HeteroOS-LRU

The limited capacity of faster memories and prioritizing page allocation from all the subsystem leads to contention. Contention exists for homogeneous memory systems too when DRAM capacity is limited, and OSes such as Linux employ LRU techniques. Briefly, Linux uses an approximate split LRU that maintains an active list of hot or recently used pages, and an inactive list with cold pages for each memory zone. Although at a high-level, the same mechanism can be adapted for heterogeneous memory, several limitations must be addressed. First, current swap-based LRUs mainly target the I/O pages [9]. Second, they use the whole system memory pressure and not the memory pressure of individual memory types as a trigger for eviction. Finally, they use a lazy approach of LRU scan and eviction only after a usage threshold is reached which can trigger a storm of allocation misses and evictions for a limited capacity FastMem. To address these issues, we design HeteroOS-LRU.

HeteroOS-LRU. First, HeteroOS extends the existing Linux page-replacement with support for memory type-specific thresholds for

triggering replacement. Second, unlike the lazy approach, HeteroOS-LRU actively monitors the active to an inactive state change of heap, I/O page cache, and slab pages and immediately evicts them from FastMem. To further reduce FastMem allocation misses, HeteroOS-LRU implements the following memory type-specific threshold. (1) During an unmap operation, several continuous pages in a VMA region are released. HeteroOS-LRU marks these pages inactive and aggressively migrates them to SlowMem. (2) I/O page and buffer cache pages are released after an I/O request, are marked inactive and immediately evicted from FastMem.

4 COORDINATED MANAGEMENT

When the on-demand allocation and LRU mechanism is not sufficient to locate free FastMem pages, the guest-OS delegates the page hotness-tracking to the VMM and also guides it to track only relevant pages. This limits the overheads of the hotness-tracking operations. However, the actual migrations are performed in the guest-OS – a fundamental difference compared to the VMM-exclusive HeteroVisor approach described in Section 2. The VMM also performs system-wide heterogeneous memory resource sharing across guest-VMs.

4.1 HeteroOS-coordinated design

HeteroOS's coordinated management (HeteroOS-coordinated) reuses HeteroVisor's hotness-tracking implementation but modifies it substantially to support a new guest-OS interface for coordination with the VMM via a split guest-OS front-end and VMM back-end model, as shown in the steps 4 to 9 in Figure 5.

OS-guided VMM-level hotness-tracking. Tracking the entire guest-VM's memory for hotness is expensive. Hence, our coordinated management reduces the scope and cost by using the guest-OS information to guide the VMM about which pages to track, and when to track.

Guiding what to track using OS-level information. The guest-OS exports a *tracking list* and an *exception list* to the VMM using a shared memory channel. The *tracking list* contains address ranges of contiguous memory regions that the VMM should track for hotness. We extract it using the virtual memory area (VMA) structure [22]. Next, tracking the short-lived I/O page cache and buffer cache pages for hotness is not useful and only adds additional overhead. Hence such pages are added to the exception list, and HeteroOS-LRU aggressively evicts them after the I/O request. Page migration of linearly mapped physically addressed page table and DMA pages is complicated and not supported by OSes such as Linux. Hence they are also added to the exception list.

Guiding when to track using architectural hints. Current systems lack the hardware for page-level hotness-tracking. The software-based methods of forcefully setting and resetting the PTE and scanning the page table can detect the number of page accesses, but lack the information about how many accesses to a page are cache hits or misses. This information is critical because migrating hot pages during an application phase with high page reuse and low processor cache miss will have limited gains from FastMem given the cost of migration. Without the additional hardware support for

Algorithm 1 HeteroOS DRF algorithm

1: R = $\{r_1, ..., r_m\}$ ▷ total memory capacities
2: C = $\{c_1, ..., c_m\}$ ▷ used memory capacities initially 0
3: $s_i (i = 1..n)$ ▷ guest i dominant shares, initially 0
4: $VM_i = \{vm_{i,1}, ..., vm_{i,m}\} (i = 1...n)$ ▷ memory resources given to guest i
5: pick guest i with lowest dominant share s_i in queue
6: D_i - guest i's memory allocation request
7: **if** $C + D_i \leqslant R$ **then**
8: $C = C + D_i$ ▷ update consumed vector
9: $VM_i = VM_i + D_i$ ▷ update $i's$ allocation vector
10: $s_i = max_{j=1}^{m}\{vm_{i,j}/r_j\}$
11: **else**
12: reclaim guest i's overcommit pages for VM_i

tracking cache hits and misses, HeteroOS monitors the LLC misses exported by the VMM in each epoch and dynamically varies the hotness-tracking and migration interval. When the LLC misses are high, the migration and tracking intervals are shorter, and when low, the interval is longer. Equation 1 shows a simple but an effective model where i and $i-1$ represent the current and previous intervals.

$$\begin{aligned} \Delta LLCMiss &= (LLCMiss_i - LLCMiss_{i-1})/LLCMiss_{i-1} \\ Interval &= Interval - (\Delta LLCMiss * Interval) \end{aligned} \tag{1}$$

Guest-OS-controlled migration. In HeteroOS, the VMM's hotness-tracking is exported to the guest-OS, and the guest-OS performs the page migrations for the following reasons.

Page state: Before a page is migrated, the OS during a page walk validates if the page is mapped to a process, and not marked for deletion or dirty (for I/O page caches). Specifically, these checks are important for most capacity-intensive applications (e.g., Graphchi) that frequently allocate (map) and de-allocate (unmap) pages. The VMM without application information can migrate pages marked for deletion only polluting FastMem. Further, it does not distinguish short-lived dirty I/O, buffer and page cache pages and migrates them only adding migration-related performance overhead.

Scalability via adaptive migration: With an increasing number of guest-VMs on a single host and the increasing application working set size, we observed that performing both hotness-tracking and migration induces significant VMM-level data structure synchronization bottlenecks even for two guest-VMs. In contrast, the guest-level migration provides the flexibility to use the application information at the OS-level to selectively migrate performance critical pages only.

4.2 Resource management with DRF

Effective resource sharing across multiple VMs is one of the primary responsibilities of the VMM. Most VMMs today employ simple but effective max-min fairness-based resource management. With max-min, the resources are first allocated based on the demands of the VMs to guarantee that each VM receives its basic share (or what it paid for). Any unused memory is evenly distributed among VMs demanding more than the fair share (overcommit). Finally, the additional memory allocated to a guest-VM is reclaimed using well-known memory ballooning [62]. However, the current mechanisms have two major limitations, (1) single resource max-min cannot

guarantee fairness for multiple memory types, and (2) the ballooning mechanisms are designed for homogeneous memory.

To address the limitations of single resource min-max fairness, we treat each memory type as a resource and extend the 'Dominant Resource Fairness' proposed by Ghodsi et al. [19] for addressing max-min fairness across multiple resources. DRF first computes the share of each resource allocated to a guest-VM. Then, as shown in Algorithm 1, a resource with a maximum share of all resources for a guest-VM is its dominant resource with a dominant share. When one or more allocation requests are made for the same resource by different VMs, DRF prioritizes allocation to a VM in the order of smallest dominant share value. Each guest-VM specifies a resource allocation vector <FastMem.pages, SlowMem.pages> during the boot. A problem with this basic model is that, when the FastMem capacity ratio is small, most VMs will always have SlowMem as the dominant resource. To address this problem, we assign weights for calculating the dominant share with resource vector specified as <FastMem.weight * FastMem.pages, SlowMem.Weight * SlowMem.pages>. For our evaluation, we use static weights ('1' for SlowMem, and '2' for FastMem). Dynamic weights can be used based on the cost($) of the resource in the datacenter. When the demand for one resource is high, then DRF reduces to a single resource max-min fairness.

Extending ballooning. For extending the ballooning mechanism and supporting memory overcommit for memory types, we provide support for guest-VMs to specify a memory type-specific minimum capacity that is reserved during the boot, and a maximum capacity that can be dynamically allocated by extracting (ballooning) from other VMs, if not over-committed. Cloud providers can directly tie a cost model to the minimum and maximum value which is beyond the paper's scope. Next, we extend the balloon drivers to support a memory type-specific balloon to inflate or shrink memory from each type. In HeteroOS, balloon drivers first use HeteroOS-LRU to find inactive pages, and if not, swap pages to the disk.

4.3 Limitations and design discussion

In HeteroOS, rather than completely reinventing the OS, we aim to utilize the virtues of existing OS memory management and extend it to support an application-transparent memory heterogeneity management.

First, HeteroOS is designed for a generic fast and slow memory type devices. Although our current OS and VMM data structures support multiple memory types, additional data placement policies, and technology specific extensions are required. For example, our OS support can be extended to provide different page allocation policies based on the latency, bandwidth, endurance and capacity of memory types. Further, we currently support only aggressive FastMem eviction policy. For multi-level memories, enabling page-type specific promotion/demotion policies can be important. For example, inactive heap pages can be demoted one level at a time (e.g., FastMem− >MediumMem− >SlowMem) because of high reuse, whereas IO buffers are mostly unused after IO completion, and can be demoted to large-but-slowest memory. We plan to focus on these extensions in our future work.

Mechanisms	Description
Heap-OD	On-demand heap allocation
Heap-IO-Slab-OD	Heap-OD + IO page cache allocation + slab allocation
HeteroOS-LRU	Heap-IO-Slab-OD + HeteroOS-LRU
HeteroOS-cordinated	HeteroOS-LRU + OS guided hotness-tracking + architecure hints

Table 5: HeteroOS incremental mechanisms.

Second, HeteroOS can only manage memory devices that are exposed to the software. Therefore, for memory technologies such as MCDRAM [58], HeteroOS can manage the software-exposed 'flat' or 'hybrid' mode, with the hardware managing the 'cache' mode. Further, the technology specific management can be added to HeteroOS by extending the NUMA policy.

Third, memory technologies such as NVM have substantial read-write latency imbalance. Our page placement and the migration policies can be extended to migrate hot and write-heavy SlowMem (NVM) pages to FastMem retaining the read-heavy pages in SlowMem. One software approach for tracking the write activity of a page is by periodically setting and resetting the write bit (PAGE_RW) of page table entries and maintaining the history similar to hotness-tracking discussed in this paper. However, this approach is approximate and can add significant software overhead. Tracking such information in the hardware and exposing it to the software (e.g., at the memory controller level [43]) can significantly reduce such overheads.

Fourth, we use DRF because it is proven to guarantee strategy-proofness (no benefits from lying) and Pareto efficiency. For guest-VMs lying about FastMem (or SlowMem) requirement, its dominant ratio increases forcing HeteroOS's ballooning to reclaim pages. For stronger security model against a malicious guest-VM, DRF can be complemented with other cloud resource security models [17, 64].

Finally, although HeteroOS is currently implemented targeting virtualized datacenters, most of the placement and management is done at the OS. Hence it can be easily applied to non-virtualized systems with bare-metal OS by just moving the page hotness-tracking and DRF into the OS.

5 EVALUATION

We evaluate HeteroOS using micro-benchmarks and real-world cloud applications and aim to answer the following questions.

- What are the implications of guest-OS heterogeneous memory awareness?
- How effective is the HeteroOS's application-transparent memory placement?
- What are the performance benefits of the coordinated guest-OS-VMM management?
- How effective is HeteroOS's DRF-based resource sharing mechanisms?

5.1 Methodology and baselines

We use a 16-core Intel Xeon 2.67 GHz dual socket system, with 16GB memory per socket. As discussed earlier in Section 2, we consider generic FastMem and SlowMem types. For FastMem, we use DRAM, and for SlowMem, we apply DRAM thermal throttling to decrease the bandwidth by ~9x and increase the latency by ~5x

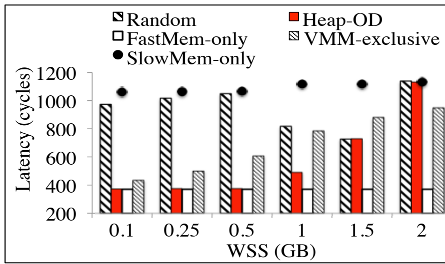

Figure 6: Memory latency benchmark.

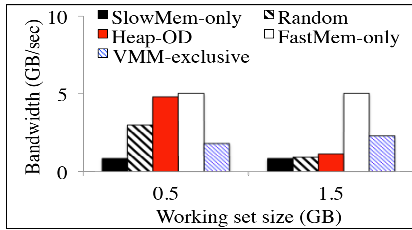

Figure 7: Stream bandwidth benchmark.

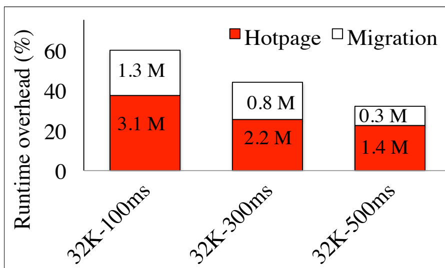

Figure 8: VMM-exclusive [24] hotness-tracking and migration cost for Graphchi (values in bars show millions of pages).

Batch size	T_{page_move} (in µs)	T_{page_walk} (in µs)
8K	25.5	43.21
64K	15.7	26.32
128K	11.12	10.25

Table 6: Per-page migration (page walk + page copy cost).

based on the industrial projections [4, 59]. A similar approach has been used by prior heterogeneous memory research [11, 24]. Each guest-VM in our evaluation has 8GB SlowMem, and we vary the FastMem capacity from 256MB to 4GB. For our evaluation, we use popular cloud applications listed in Table 2 with varying CPU, memory, storage, and network intensity. We use two baselines, (1) SlowMem-only – a naive approach always using SlowMem memory, and (2) FastMem-only – an ideal approach in which application memory requirements are always satisfied by FastMem with unlimited capacity. Table 5, summarizes the incremental HeteroOS approaches.

5.2 Micro benchmark analysis

We next evaluate the implications of different heterogeneous memory placement and management methods using microbenchmarks.

Memory latency and bandwidth. We first perform analysis using the memory latency microbenchmark – 'memlat' [60] in Figure 6, and a well-known memory bandwidth Stream benchmark in Figure 7. We limit the FastMem capacity to 0.5GB and SlowMem to 3.5GB. We compare (1) HeteroOS's simple heap-only allocation (Heap-OD) against (2) the VMM-exclusive migration-only approach, (3) Random - an approach that randomly allocates and places pages in the FastMem reserved at the boot time without heterogeneity-awareness, and the baselines (4) SlowMem-only and (5) FastMem-only. The benchmarks allocate only heap pages. For the latency benchmark, we vary the working set size from 0.25GB to 2GB in the x-axis with average latency in the y-axis. For the bandwidth (Stream) benchmark, we show results for 0.5GB and 1.GB, and the y-axis shows the corresponding memory bandwidth.

Observations. First, the Random approach shows a non-deterministic behavior for both the latency and the bandwidth benchmarks. When the working set is smaller (0.25GB) than the available FastMem, the latency is high. However, increasing the working set (0.5 and 1 GB) increases memory pressure forcing frequent FastMem allocations resulting in a lower latency. Increasing the working set further shows significant degradation. Next, with Heap-OD, for working set smaller than FastMem, on-demand allocation shows ideal latency and bandwidth comparable to the FastMem-only approach. Increasing the working set beyond 0.5GB results in a gradual increase in latency and reduction in the bandwidth. In contrast, the VMM-exclusive approach, by relying upon migration for even a smaller working set shows the highest latency and the lowest bandwidth. However, it is important to note that when the working set is increased beyond the capacity of FastMem (0.5GB), the VMM-exclusive method, by hotness-tracking and evicting least used FastMem pages, always achieves a zero FastMem allocation miss ratio. The results highlight that *First, On-demand allocation is important when the working set is less than FastMem capacity. Second, for a larger working set, hotness-tracking and migration are essential.*

Hotness tracking and migration cost. Next, to understand the cost of hotness-tracking and page migration, we use the state-of-the-art VMM-exclusive implementation, HeteroVisor, and enable hotness-tracking and migration for Graphchi [34] application. Our goal is to understand the software overheads; hence we do not emulate NVM bandwidth and latency. We vary the tracking hotness-tracking intervals from 100ms-500ms for every 32K pages similar to HeteroVisor [24]. The x-axis shows the hotness-tracking intervals (100ms-500ms) for scanning 32K pages of a VM and the y-axis shows the runtime overhead (in %). Clearly, even with a 500ms interval, the migration, and hotness-tracking adds up to 32% overhead on the application, and for 100ms, the overheads increase by up to 60%. Hot page scan requires frequent TLB invalidation to force the hardware to set a page table access bit upon reference. Hence, hotness-tracking is even more expensive compared to the migrations.

Regarding the actual page migration cost, Table 6 shows the average per-page page copy cost (T_{page_move}) and the page table walk cost (T_{page_walk}). We vary the batch size of such page movement. It is interesting to note that cost of page walk is even more expensive than actual migration. Batching the page walks and copy operations reduces the average cost by reducing the page tree traversal cost of higher memory bandwidth usage. *Overall, the results show that relying exclusively on migrations for heterogeneous memory management is highly suboptimal.*

5.3 Guest-OS memory placement

We next evaluate the effectiveness of HeteroOS in leveraging application page use information for right memory placement inside the guest-OS. Figure 9 shows the results for all applications listed in Table 2. We compare the following approaches summarized in Table 5: (1) Heap-OD, (2) Heap-IO-Slab-OD, (3) HeteroOS-LRU, (4) NUMA-preferred – the existing Linux's preferred NUMA node [33] policy by enabling our guest-OS heterogeneity awareness, and finally, (5) FastMem-only - shown with the dashed line. We do not discuss NGinx [44] because it has less than 10% impact from heterogeneous memory, as discussed in Section 2.The y-axis shows the performance gain percentage compared to the naive SlowMem-only approach. To measure the effectiveness of each of the approaches in allocating FastMem pages, Figure 10 shows the miss ratio of total FastMem page allocation misses to the total allocation requests.

Observations. First, making the guest-OS heterogeneity-aware and providing on-demand memory allocation provides significant benefits. When prioritizing only the heap pages with Heap-OD (solid gray bars in the figure), the heap-intensive Graphchi and Metis show 121% and 84% gains even with 1/2 FastMem capacity ratio. Applications such as Graphchi that frequently allocate-deallocate memory and show up to 2x gains even with 1/4 ratio. This is in contrast to Metis's 45% gains as it seldom releases memory and has a large 5GB working set size. X-Stream, LevelDB, and Redis show limited gains with Heap-only prioritization. Both LevelDB and X-Stream are highly page-cache intensive and prioritizing the I/O page cache along with the heap is of key importance. For LevelDB, placing buffer cache pages in FastMem speeds up logging and read operations via a memory-mapped database. This results in a 2x increase in the overall throughput. X-Stream computes over a memory mapped I/O data, and FastMem-based page-cache alone reduces the runtime by 50% with an overall 2x reduction with Heap-IO-Slab-OD for 1/2 capacity ratio and ~80% for 1/4 ratio. For Redis, prioritizing the slab allocations pages used for the network send and receive buffers in FastMem significantly improves throughput.

Finally, NUMA-preferred bars represent the implications of just using the existing NUMA management for dealing with heterogeneous memories by setting FastMem as preferred node. Applications CPUs first use FastMem, and when free pages are exhausted, SlowMem is used. We use NUMA-preferred placement for comparison because we notice a significant slowdown with other policies such as 'local node first' or the Linux automatic NUMA balancing [23] policy because some cores are bounded to SlowMem even when FastMem is available, resulting in a significant slowdown. The results in the figure demonstrate that existing NUMA policies can

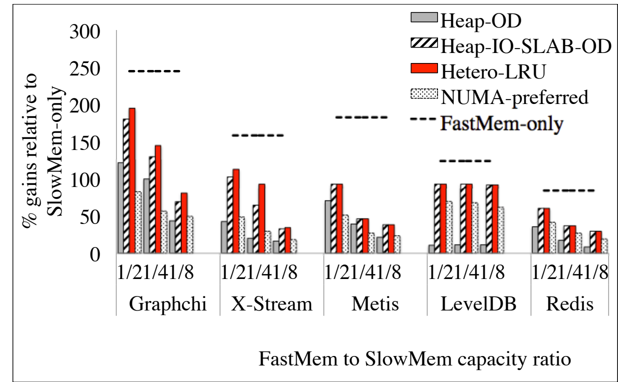

Figure 9: Impact of OS heterogeneity awareness. Y-axis shows gains (%) relative to using only SlowMem.

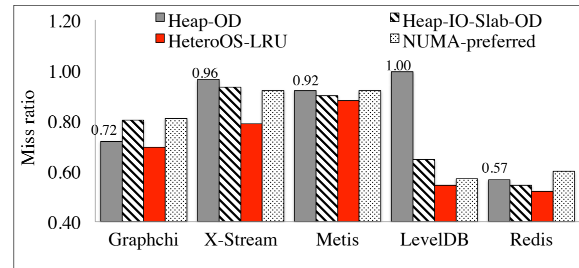

Figure 10: FastMem allocation miss ratio for 1/8 FastMem capacity ratio.

extract some benefits when FastMem capacity is high (e.g., 82% for Graphchi with 1/2 capacity ratio compared to SlowMem only approach). But for lower FastMem capacity, the benefits significantly degrade compared to all the HeteroOS approaches, and for all applications and configurations. This is because existing NUMA mechanisms (a) fail to differentiate the significant latency and bandwidth differences, (b) always prioritize heap and lack smart memory placement, (c) lack active contention resolution, and finally, (d) lack guest-OS-VMM coordinated management discussed shortly.

The results validate the importance of guest-OS heterogeneity-awareness, smart memory placement, and HeteroOS-LRU principles for in-memory, storage, and network-intensive applications.

5.4 Impact of coordinated management

We next evaluate the impact of HeteroOS's VMM-guest coordinated management using the same applications as before for 1/4 and 1/8 capacity configurations. In Figure 11, the y-axis shows the speedup relative to the SlowMem-only baseline, and the dotted lines represents the best case FastMem-only approach. Figure 12 compares the gains only from migrations and the total pages migrated (in millions) for three applications (for brevity) relative to the Heap-IO-Slab-OD which completely relies on smart page placement without any migration. In Figure 11, we compare the following approaches described in Table 5 – (1) VMM-exclusive with hot page scan of 16K guest-VM pages in a 100msec interval, (2) HeteroOS-LRU - the best case approach analyzed in Figure 9, and (3) the HeteroOS-coordinated where the guest guides the VMM to scan only relevant application

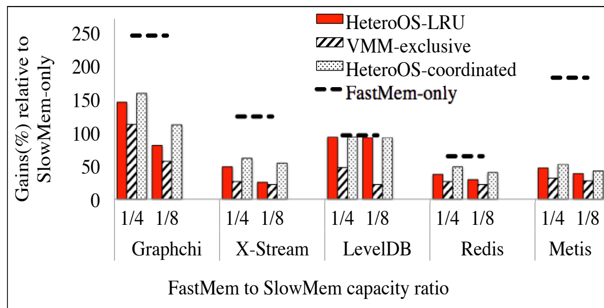

Figure 11: Impact of HeteroOS-coordinated.

Apps.	VMM-exclusive	HeteroOS-LRU	HeteroOS-Coordinated
Graphchi	-30.0 (0.69)	10.0 (0.10)	40.0 (0.33)
Redis	-20.0 (0.51)	2.1 (0.11)	19.0 (0.26)
LevelDB	-10.0 (0.14)	20.0 (0.01)	20.0 (0.08)

Figure 12: Gains exclusively from page migrations relative to Heap-IO-Slab-OD. Values in brackets shows total migration in millions.

pages instead of the entire guest-VM, and to dynamically vary the hotness scanning interval from 50ms to 1 second based on the increase in cache misses using Equation 1 in Section 4.

VMM-exclusive vs. HeteroOS-LRU. The VMM-exclusive migration-only approach performs poorly due to lazy FastMem page allocation even when FastMem pages are free, and lack of information about FastMem pages that are already released by application for OS garbage collection resulting in a ~40-45% lower page reuse compared to HeteroOS-LRU. Additionally, applications such as LevelDB with a small working set size that fits in FastMem show less than 10% gains with the VMM-exclusive approach. The VMM-exclusive is useful for long-running and heap-intensive applications such as Metis where its gains are comparable with HeteroOS-LRU.

HeteroOS-LRU vs. HeteroOS-coordinated. The coordinated approach combines HeteroOS-LRU and VMM's hotness-tracking and also reduces hotness-tracking overheads with guest-OS hints and the migration overheads by monitoring cache miss counters. As a result, the HeteroOS-coordinated approach outperforms the guestOS-only HeteroOS-LRU management approach for both memory capacity and cache-intensive applications. For Graphchi and X-Stream, HeteroOS-coordinated approach improves the gains by 28%, and 16% compared to HeteroOS-LRU for 1/4 capacity ratio. For LevelDB, with a small working set size, VMM-level hotness-tracking with HeteroOS-coordinated does not add much to the HeteroOS-LRU's gains. As shown in Table 12, while the page migrations with HeteroOS-coordinated increase relative to the eviction and migrations by HeteroOS-LRU, HeteroOS-coordinated also improves application performance validating the need for a coordinated approach.

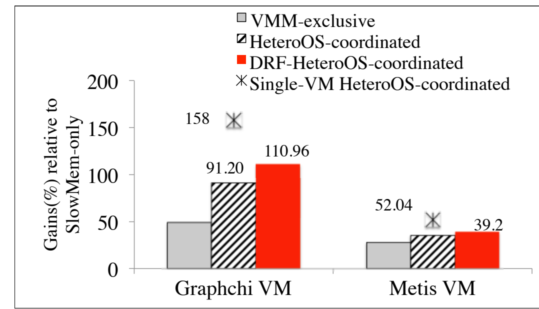

Figure 13: Impact of multi-VM resource sharing.

5.5 Weighted DRF-based resource sharing

To understand the impact of HeteroOS's DRF-based heterogeneous memory sharing across VMs, we compare DRF-based HeteroOS-coordinated with the single resource max-min fairness-based HeteroOS-coordinated and the VMM-exclusive approach. We run a Graphchi VM and Metis VM on a system with total 4GB FastMem and 8GB SlowMem. For Graphchi, we use a Twitter dataset that requires 6GB of total heap capacity with an active working set size of just 1.5GB. Hence, the Graphchi VM is reserved with a 1GB FastMem, and 4GB SlowMem with a resource vector $< 2*1GB, 1*4GB >$ where 2 and 1 represent the weight of FastMem and SlowMem resource respectively. For Metis, our dataset uses 8GB of the heap and has a working set size of 5.4GB. Hence, we use a $< 2*3GB, 1*4GB >$ configuration. When using DRF, for the Graphchi VM, SlowMem (1*4GB) is the dominant resource, and for the Metis VM, FastMem (2*3GB) is dominant. Note that the existing max-min mechanisms can guarantee fairness for only one memory type. In Figure 13, the bars represent the multi-VM execution time, whereas the stars indicate the single-VM HeteroOS-coordinated time (best case in the earlier evaluation).

Max-min VMM-exclusive vs. HeteroOS-coordinated. First, as expected, resource contention across multiple VMs (Graphchi and Metis) slows down the performance compared to a single VM HeteroOS-coordinated approach. Next, although the simple max-min fairness-based HeteroOS-coordinated outperforms the VMM-exclusive approach by reducing migrations, however, the Graphchi VM suffers ~73% slowdown relative to the single-VM execution baseline due to resource contention with Metis VM which only suffers 36% slowdown. The memory-hungry Metis first exhausts the reserved FastMem and then starts exhausting SlowMem by ballooning out the Graphchi VM's SlowMem pages too. This happens because *a single resource max-min can guarantee fairness of only one resource (FastMem in this case).*

Max-min vs. Weighted DRF HeteroOS-coordinated. Unlike the single resource max-min fairness, DRF by defining a dominant resource – FastMem for Metis VM, SlowMem for Graphchi VM, guarantees the 4GB SlowMem availability for Graphchi VM. As a result, DRF-based HeteroOS-coordinated improves the Graphchi VM's performance by 42%, and 87% compared to a simple max-min fairness-based HeteroOS-coordinated and VMM-exclusive approach,

respectively. The overall system performance also improves.

Summary. First, by making guest-VMs heterogeneous memory-aware, and extracting the heap, IO, and network page use information for smart memory placement, HeteroOS provides up to 180% gains over the naive SlowMem-only approach. HeteroOS-LRU reduces contention and increases FastMem use to improve the gains by 194% (~3x). Next, the HeteroOS-coordinated exploits the OS-level information to guide the VMM's hotness-tracking and provides up to 2x gains over the VMM-exclusive approach. Finally, the DRF's multi-resource fairness-based sharing provides up to 87% gains compared to the VMM-exclusive approach.

6 RELATED WORK

Prior research has dealt with heterogeneous memory at the hardware, systems software, or the application-level.

Hardware support. Several hardware efforts have explored the use of byte addressable NVMs such as phase change memory (PCM) and on-chip 3D-DRAM. Some have focussed specifically on the persistence aspect of NVM [36, 40, 49, 51] and others have mainly explored the benefits of using NVM for additional capacity. For stacked 3D-DRAMs, some researchers have considered using them as a large last level (L4) cache [6, 29, 41, 48, 66]. In contrast, other research [2, 7, 13, 25, 43, 46] use them as a high bandwidth DRAM due to significant hardware changes (cache controller and tag space changes), and lack of application flexibility.

Hardware-based management. Batman [7] modifies the memory controller to randomize data placement for increasing the cumulative DRAM and stacked 3D-DRAM bandwidth. Meswani et al. [43] discuss extending the TLB and the memory controller with additional logic for identifying page hotness. To reduce page migration cost, X.Dong et al. [13] propose SSD FTL-like mapping [8] that can map FastMem slots with a physical address dynamically. M.Oskin et al. [45] propose an architectural mechanism to selectively invalidate entries in the TLB for reducing the TLB shoot-downs during migrations. Ramos et al. [52] propose a hybrid design with hardware-driven page placement policy and the OS periodically updating its page tables using the information from the memory controller. In contrast to all these solutions, HeteroOS is an OS-level solution without hardware changes but can complement prior hardware-level proposals.

OS and software-level management. Prior software solutions mostly rely on application-level extensions with new interfaces [37] and offline memory classification [15]. Phadke et al. [46] categorize application data structures into latency, bandwidth or CPU-intensive to guide the OS-level page allocation. Dulloor et al. [15] propose X-mem that uses static analysis information to guide the user-level library allocator. HeteroOS does not require any static analysis or application-level changes. Most prior studies target in-memory applications only [15, 43, 46], whereas the HeteroOS design addresses in memory, storage, and network-intensive applications. HetroOS is the first system to manage memory heterogeneity for both non-virtualized and virtualized systems. Unlike prior solutions that only discuss the differences between homogeneous and heterogeneous NUMA systems, HeteroOS also shows how to extend NUMA-based abstraction for efficient management of heterogeneous memory.

7 CONCLUSION

In this paper, we study the impact of memory heterogeneity on datacenter applications and address the inefficiency of existing homogeneous memory management and page migration-based techniques. We design an application-transparent OS- and VMM-level solution, HeteroOS, which provides guest-OS with heterogeneity awareness and extracts rich OS-level information to provide smart memory placement reducing page migrations. Furthermore, it combines the guest-OS information with the VMM's privileged hardware control to coordinate placement of performance-critical pages to the right memory and resource fairness across VMs using a novel DRF mechanism. Overall results show up to 2x benefits over state-of-the-art approaches. We believe our analysis of software overheads such as hotness-tracking, page movement cost, and design methods will benefit hardware architects and OS developers to design more optimal methods and technology-specific solutions for managing heterogeneous memory.

ACKNOWLEDGMENTS

We would like to thank all of the anonymous reviewers for their valuable feedback. Dr. Greg Eisenhauer from Georgia Tech provided valuable suggestions on earlier drafts of this paper. We would also like to thank the U.S. Department of Energy and Intel Labs for their funding and access to the Intel NVM emulator platform.

REFERENCES

[1] Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Onyx: a protoype phase change memory storage array. In *HotStorage '11*.

[2] Berkin Akin, Franz Franchetti, and James C. Hoe. 2015. Data Reorganization in Memory Using 3D-stacked DRAM. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 131–143. https://doi.org/10.1145/2749469.2750397

[3] Oren Avissar, Rajeev Barua, and Dave Stewart. 2002. An Optimal Memory Allocation Scheme for Scratch-pad-based Embedded Systems. *ACM Trans. Embed. Comput. Syst.* 1, 1 (Nov. 2002), 6–26. https://doi.org/10.1145/581888.581891

[4] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H. Loh, Don McCaule, Pat Morrow, Donald W. Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. 2006. Die Stacking (3D) Microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, USA, 469–479. https://doi.org/10.1109/MICRO.2006.18

[5] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. 2008. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 43–57. http://dl.acm.org/citation.cfm?id=1855741.1855745

[6] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2014. CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/MICRO.2014.63

[7] Chia-Chen Chou, Aamer Jaleel, and Moinuddin Qureshi. 2015. BATMAN: Maximizing Bandwidth Utilization for Hybrid Memory Systems. In *Technical Report, TR-CARET-2015-01 (March 9, 2015)*.

[8] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. 2009. A Survey of Flash Translation Layer. *J. Syst. Archit.* 55, 5-6 (May 2009), 332–343. https://doi.org/10.1016/j.sysarc.2009.03.005

[9] Jonathan Corbet. 2016. Linux Swap priority. https://lwn.net/Articles/690079. (2016).

[10] Jonathan Crobett. 2003. Linux object-based reverse-mapping. https://lwn.net/Articles/23732/. (2003).

[11] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. 2011. MemScale: Active Low-power Modes for Main Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 225–238. https://doi.org/10.1145/1950365.1950392

[12] Peter J. Denning. 1968. The Working Set Model for Program Behavior. *Commun. ACM* 11, 5 (May 1968), 323–333. https://doi.org/10.1145/363095.363141

[13] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. 2010. Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/SC.2010.50

[14] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/2592798.2592814

[15] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 15, 16 pages. https://doi.org/10.1145/2901318.2901344

[16] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2150976.2150982

[17] Tal Garfinkel and Mendel Rosenblum. 2005. When Virtual is Harder Than Real: Security Challenges in Virtual Machine Based Computing Environments. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10 (HOTOS'05)*. USENIX Association, Berkeley, CA, USA, 20–20. http://dl.acm.org/citation.cfm?id=1251123.1251143

[18] Sanjay Ghemawat and Jeff Dean. 2011. Google LevelDB . http://tinyurl.com/osqd7c8. (2011).

[19] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, USA, 323–336. http://dl.acm.org/citation.cfm?id=1972457.1972490

[20] Jerome Glisse. 2016. Linux heterogeneous memory management. https://lwn.net/Articles/679300/. (2016).

[21] Maya Gokhale, Scott Lloyd, and Chris Macaraeg. 2015. Hybrid Memory Cube Performance Characterization on Data-centric Workloads. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms (IA3 '15)*. ACM, New York, NY, USA, Article 7, 8 pages. https://doi.org/10.1145/2833179.2833184

[22] Mel Gorman. 2004. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[23] Mel Gorman. 2012. Foundation for automatic NUMA balancing. https://lwn.net/Articles/523065. (2012).

[24] Vishal Gupta, Min Lee, and Karsten Schwan. 2015. HeteroVisor: Exploiting Resource Heterogeneity to Enhance the Elasticity of Cloud Platforms. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '15)*. ACM, New York, NY, USA, 79–92. https://doi.org/10.1145/2731186.2731191

[25] Anthony Gutierrez, Michael Cieslak, Bharan Giridhar, Ronald G. Dreslinski, Luis Ceze, and Trevor Mudge. 2014. Integrated 3D-stacked Server Designs for Increasing Physical Density of Key-value Stores. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 485–498. https://doi.org/10.1145/2541940.2541951

[26] Heather Hanson and Karthick Rajamani. 2012. What Computer Architects Need to Know About Memory Throttling. In *Proceedings of the 2010 International Conference on Computer Architecture (ISCA'10)*. Springer-Verlag, Berlin, Heidelberg, 233–242. https://doi.org/10.1007/978-3-642-24322-6_20

[27] Jingtong Hu, Qingfeng Zhuge, Chun Jason Xue, Wei-Che Tseng, and Edwin H.-M. Sha. 2013. Software Enabled Wear-leveling for Hybrid PCM Main Memory on Embedded Systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '13)*. EDA Consortium, San Jose, CA, USA, 599–602. http://dl.acm.org/citation.cfm?id=2485288.2485434

[28] Sysoev Igor. 2004. NGinx Webserver. http://nginx.org. (2004).

[29] Xiaowei Jiang, N. Madan, Li Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, D. Solihin, and R. Balasubramonian. 2010. CHOP: Adaptive filter-based DRAM caching for CMP server platforms. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. 1–12. https://doi.org/10.1109/HPCA.2010.5416642

[30] Crobett Jonathan. 2012. Linux Swapping. https://lwn.net/Articles/495543. (2012).

[31] Yongsoo Joo, Dimin Niu, Xiangyu Dong, Guangyu Sun, Naehyuck Chang, and Yuan Xie. 2010. Energy- and Endurance-aware Design of Phase Change Memory Caches. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 136–141. http://dl.acm.org/citation.cfm?id=1870926.1870961

[32] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. 2016. pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 13, 16 pages. https://doi.org/10.1145/2901318.2901325

[33] Michael Kerrisk. 2007. Linux NUMA policies. http://man7.org/linux/man-pages/man3/numa.3.html. (2007).

[34] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 31–46. http://dl.acm.org/citation.cfm?id=2387880.2387884

[35] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and others. Architecting phase change memory as a scalable dram alternative. In *ISCA '09*.

[36] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *ISCA*. ACM.

[37] Felix Xiaozhu Lin and Xu Liu. 2016. Memif: Towards Programming Heterogeneous Memory Asynchronously. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 369–383. https://doi.org/10.1145/2872362.2872401

[38] Duo Liu, Tianzheng Wang, Yi Wang, Zhiwei Qin, and Zili Shao. 2011. PCM-FTL: A Write-Activity-Aware NAND Flash Memory Management Scheme for PCM-Based Embedded Systems. In *Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium (RTSS '11)*. IEEE Computer Society, Washington, DC, USA, 357–366. https://doi.org/10.1109/RTSS.2011.40

[39] Duo Liu, Tianzheng Wang, Yi Wang, Zhiwei Qin, and Zili Shao. 2012. A Block-level Flash Memory Management Scheme for Reducing Write Activities in PCM-based Embedded Systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '12)*. EDA Consortium, San Jose, CA, USA, 1447–1450. http://dl.acm.org/citation.cfm?id=2492708.2493062

[40] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. 2014. NVM Duet: Unified Working Memory and Persistent Store Architecture. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 455–470. https://doi.org/10.1145/2541940.2541957

[41] Gabriel Loh and Mark D. Hill. 2012. Supporting Very Large DRAM Caches with Compound-Access Scheduling and MissMap. *IEEE Micro* 32, 3 (May 2012), 70–78. https://doi.org/10.1109/MM.2012.25

[42] Sally A. McKee. 2004. Reflections on the Memory Wall. In *Proceedings of the 1st Conference on Computing Frontiers (CF '04)*. ACM, New York, NY, USA, 162–. https://doi.org/10.1145/977091.977115

[43] M.R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G.H. Loh. 2015. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. 126–136. https://doi.org/10.1109/HPCA.2015.7056027

[44] Rick Nelson. 2014. NGinx memory usage. https://www.nginx.com/blog/nginx-websockets-performance/. (2014).

[45] Mark Oskin and Gabriel H. Loh. 2015. A Software-Managed Approach to Die-Stacked DRAM. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 188–200. https://doi.org/10.1109/PACT.2015.30

[46] Sujay Phadke and S. Narayanasamy. 2011. MLP aware heterogeneous memory system. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*. 1–6. https://doi.org/10.1109/DATE.2011.5763155

[47] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 14–23. https://doi.org/10.1145/1669112.1669117

[48] Moinuddin K. Qureshi and Gabe H. Loh. 2012. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a

Simple and Practical Design. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 235–246. https://doi.org/10.1109/MICRO.2012.30

[49] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. *SIGARCH Comput. Archit. News* 37, 3 (June 2009), 24–33. https://doi.org/10.1145/1555815.1555760

[50] Milan Radulovic, Darko Zivanovic, Daniel Ruiz, Bronis R. de Supinski, Sally A. McKee, Petar Radojković, and Eduard Ayguadé. 2015. Another Trip to the Wall: How Much Will Stacked DRAM Benefit HPC?. In *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS '15)*. ACM, New York, NY, USA, 31–36. https://doi.org/10.1145/2818950.2818955

[51] L. Ramos and R. Bianchini. 2012. Exploiting Phase-Change Memory in Cooperative Caches. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*. 227–234. https://doi.org/10.1109/SBAC-PAD.2012.11

[52] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. ACM, New York, NY, USA, 85–95. https://doi.org/10.1145/1995896.1995911

[53] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. 13–24.

[54] David Rientjes. 2007. Linux Fake NUMA Patch. https://www.kernel.org/doc/Documentation/x86/x86_64/fake-numa-for-cpusets. (2007).

[55] D.A. Roberts. 2016. Reliable wear-leveling for non-volatile memory and method therefor. (May 26 2016). http://www.google.ch/patents/US20160147467 US Patent App. 14/554,972.

[56] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 472–488. https://doi.org/10.1145/2517349.2522740

[57] Salvatore Sanfilippo. 2009. Redis. http://redis.io/. (2009).

[58] Avinash Sodani. 2015. Knights landing (KNL): 2nd Generation Intel Xeon Phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*. 1–24. https://doi.org/10.1109/HOTCHIPS.2015.7477467

[59] Billy Tallis. 2017. Intel-Micron Memory 3D XPoint. goo.gl/wT4rQ6. (2017).

[60] Drepper Ulrich. 2007. "What every programmer should know about memory,". www.akkadia.org/drepper/cpumemory.pdf. (2007).

[61] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1960475.1960480

[62] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 181–194. https://doi.org/10.1145/844128.844146

[63] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News* 23, 1 (March 1995), 20–24. https://doi.org/10.1145/216585.216588

[64] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 203–216. https://doi.org/10.1145/2043556.2043576

[65] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2540708.2540744

[66] Li Zhao, R. Iyer, R. Illikkal, and D. Newell. 2007. Exploring DRAM cache architectures for CMP server platforms. In *Computer Design, 2007. ICCD 2007. 25th International Conference on*. 55–62. https://doi.org/10.1109/ICCD.2007.4601880