OS SUPPORT FOR HETEROGENEOUS MEMORY MANAGEMENT

A Thesis Proposal
Presented to
The Academic Faculty

by

Sudarsun Kannan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
November 2016

OS SUPPORT FOR HETEROGENEOUS MEMORY MANAGEMENT

Committee:

Dr. Ada Gavrilovska, Committee Chair
School of Computer Science
Georgia Institute of Technology

Professor Karsten Schwan, Advisor
School of Computer Science
Georgia Institute of Technology

Professor UmaKishore Ramachandran
School of Computer Science
Georgia Institute of Technology

Professor Sudhakar Yalamanchili
Electrical and Computer Engineering
Georgia Institute of Technology

Professor Moinuddin Qureshi
Electrical and Computer Engineering
Georgia Institute of Technology

Professor Remzi H. Arpaci-Dusseau
Department of Computer Science
University of Wisconsin-Madison

Date Approved: XXXX

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

OS Support for Heterogeneous Memory Management

Sudarsun Kannan

33 Pages

Directed by Professor Karsten Schwan, Dr. Ada Gavrilovska

To address the 'memory wall' problem of future datacenter servers, vendors are creating heterogeneous memory structures, supplementing DRAM with on-chip stacked 3D-RAM and high capacity non-volatile memory (NVM). Each of these technologies differs significantly in terms of density, bandwidth, and latency. Consequently, future systems are expected to address these requirements with heterogeneous memory architectures. However, memory heterogeneity will increase the software/OS management complexity. Therefore, a unified OS-level abstraction and efficient management principles are required for maximizing application benefits in both virtualized and non-virtualized systems. To date, operating systems, or hypervisors, do not provide a unified memory abstraction, and are ineffective in dealing with memory heterogeneity. To address these challenges, this thesis proposes an OS design 'HeteroMem' for heterogeneous memory support and makes the following contribution.

First, the thesis proposes a unified NUMA-like abstraction for different heterogeneous memory resources. A unified OS-level abstraction is not only important for seamless scaling, but also for scalable OS-level management. The abstraction is exposed to applications as well as other OS subsystems. Second, HeteroMem supports and builds customized managers specific to memory characteristics such as capacity, bandwidth, and latency, by extending the OS virtual memory (VM) subsystem. Extending the VM subsystem for all memory types enables HeteroMem to seamlessly scale across different memory, and to also inherit the virtues of VM such as efficient cache and TLB use. Third, HeteroMem exploits the OS-level information about applications for intelligent data placement with a focus on reducing data movement overheads. Fourth, HeteroMem supports memory heterogeneity in virtualized environment by exposing different memory types to virtual machines (guest-OSs), and enabling more effective coordinated hypervisor-guest OS memory management, as opposed to a hypervisor-only methods used in the state of the art systems.

Further, to exploit the persistence benefits in memory types such as NVM, this thesis extends beyond memory scaling to provide fast persistent storage in object stores, and fault tolerance mechanisms such as checkpointing. Further, supporting persistence is expensive in terms of processor cache sharing impact, and energy overheads. This thesis also investigates efficient system software principles to reduce such overheads.

CHAPTER I

INTRODUCTION

The combined rapid increase in transistor density, platform-level parallelism, and use of large datasets in modern scientific and commercial workloads is exerting severe memory pressure on server systems. However, DRAM scalability, higher access latency, and lower bandwidth continue to be a bottleneck due to energy and cost limitations. This has led researchers to the explore the use of alternative memory technologies which can supplement traditional DRAM by providing higher capacity (C), lower latency (L), and increased bandwidth (B). However, recent trends show that a single memory technology is not expected solved all the bottlenecks (C, L, and B). For instance, byte-addressable nonvolatile memories (NVMs) such as phase change memory (PCM), are expected to provide 2x-4x higher memory capacity than DRAM, but can add around 5x higher write latency, and up to 5x-8x lower bandwidth. Similarly, on-chip stacked 3D-DRAM [4, 14, 24, 29, 15] are expected to increase memory bandwidth [14] by 8x-14x, but are expected to have limited capacity (2x-4x lower capacity than DRAM). Therefore, to provide applications with large capacity, low latency, and high bandwidth, future systems will support multiple levels of memory, (i.e.,) heterogeneous memory. Figure 1 shows a high-level view of heterogeneous memory in the system stack.

Memory heterogeneity will increase the complexity of OS-level/software memory management. State of the art hardware and software research has proposed solutions that are customized for a specific memory technology. Briefly, research such as [40, 24, 31, 29, 15] have proposed using on-chip 3D-DRAM as a last-level cache, whereas other research such as [22, 14, 17, 12] have discussed the advantages of using them as a hardware and OS-managed fast memory. Similarly, prior research on NVMs can be classified into research that use application transparent methods for providing additional capacity benefits such as [32, 28, 33], or hybrid solutions that exploit non-volatility to provide fast persistence, and also additional capacity [38, 16, 33, 18].

Yet there has been little work to understand how future operating systems should manage these complex memory structures all at once: What should be the OS-level abstraction and design? How should the date be placed and when should it be moved? How to do so in the typically virtualized settings of cloud and datacenter workloads? We next discuss these questions in detail.

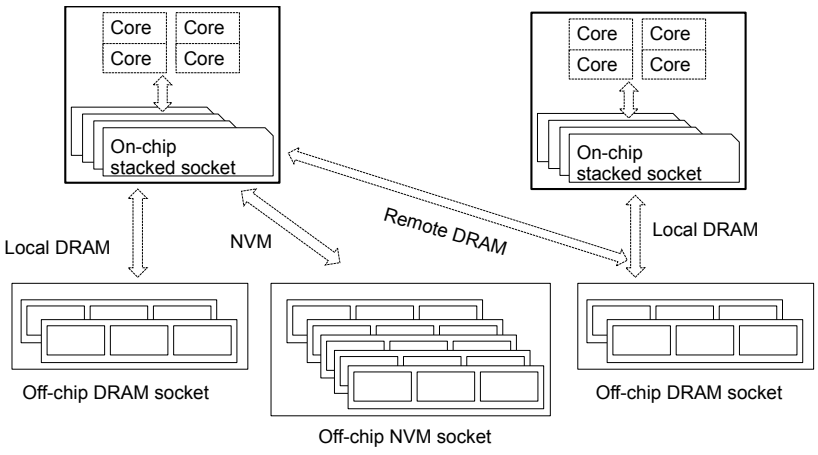


Figure 1: Heterogeneous memory system architecture

1.1 Statement of Problem

- **Need for automatic memory scaling across heterogeneous memory** Although new hardware technologies can scale memory capacity, the system software should support automatic memory scaling across these memories. Automatic scaling refers to the seamless use of one memory when other memory resources are exhausted. Although, different memory type specific solutions have been proposed, surprisingly, no system software or OS-level design exist that can automatically scale applications heap across different memory types. This thesis aims to provide automatic memory scaling across heterogeneous memory.

- **OS-level abstraction** To seamlessly scale applications heap across different memory devices, it is important to have a unified OS-level abstraction of different memory types. Additionally, a unified OS-level abstraction is not only important for seamless scaling, but also scalable OS-level management of such memories. Currently, no such abstractions exist. For example, current NVM-based solutions rely on the virtual file subsystem (VFS) to provide persistence and memory scaling [38, 16, 33, 18]. Consequently, the virtual memory (VM) subsystem cannot manage or scale across NVMs. This thesis proposes a virtual memory management of all the memory devices, with an NUMA node-like abstraction, and mechanisms that provide memory specific optimizations.

- **Flexible memory placement policies for applications** The OS should provide transparent scaling, but importantly, users/applications should have the flexibility for data placement across different memory types. Although, current solutions provide direct allocations to a specific memory type, they lack the capability to support policies (like NUMA) enables application developers for effective use of heterogeneous memory.

- **Exploiting OS information for prioritizing memory placement** Applications can provide effective memory placement, but they lack the holistic view of the system or its resource usage. Hence an efficient OS-level mechanism for efficient memory placement is critical for optimal use of heterogeneous memory, given their significant capacity, latency, and bandwidth differences. Current OS-level solutions just rely on expensive migrations neglecting the rich OS-level information about applications memory use. This thesis illustrates mechanisms to exploit such OS-level information for optimal memory placement that reduces migrations.

- **Improving the efficiency of migrations** Applications are highly dynamic with their memory usage patterns. Hence, it is not feasible to always place data right in a heterogeneous memory system. Prior research has proposed a static analysis and hotness tracking mechanisms for data migrations across memory. While static analysis for large scale dynamic applications is hard, software-based hotness tracking and migration are expensive because of data copy, page table updates, and TLB invalidation overheads. In this thesis, we show that OS-level information about the application (not to be confused with application hints) can significantly reduce hotness tracking overheads, and improve the effectiveness of migrations.

- **Cache and TLB efficiency** Efficient use of processor cache and TLB is a critical factor for application speedup. An application with all its data structure in a slower but a larger memory, but high cache efficiency can perform equally well compared to an application with all its data in a faster memory. But cache and TLB efficiency not only depends on application access pattern but also how the OS-level data structures are designed and managed. For example, prior research has proposed extensions to VFS for enabling capacity-use from NVMs. However, the VFS data structures were originally designed to support block-based disk is highly cache and TLB inefficient. Using them for providing additional capacity impose a severe performance penalty on applications. By embracing the virtues of a VM-based design for different memory devices, we show that cache and TLB efficiency can be improved significantly.

- **Support for memory heterogeneity in virtualized systems** It is a well-known fact that most large scale applications currently run in virtualized datacenters. Hence, it is obvious that the heterogeneous memory management solutions should be applicable to virtualized

environments too. Few proposals such as [22] exist for memory heterogeneity management in virtualized systems. Even in such proposal, the guest-OS and applications are transparent to heterogeneity, and all the management is done at the hypervisor level. However, a significant drawback is that the hypervisor has limited information about application resulting in an inefficient use of memory heterogeneity.

1.2 Thesis statement.

Future datacenter systems will support memory heterogeneity to address the memory wall problem and this thesis proposes an efficient OS design for heterogeneous memory management that supports seamless scaling, data placement flexibility, and intelligent OS-level data placement for achieving maximum performance benefits in virtualized and non-virtualized systems.

1.3 Contribution.

First, the thesis analyzes the impact memory heterogeneity for different classes of application in Chapter 2. Then it proposes a unified NUMA-like abstraction for different heterogeneous memory resources in Chapter 3. Second, HeteroMem supports and builds customized managers specific to memory characteristics such as capacity, bandwidth, and latency, by extending the OS virtual memory (VM) subsystem, also discussed in Chapter 3. Third, HeteroMem exploits the OS-level information about applications for intelligent data placement with a focus on reducing data movement overheads. Fourth, HeteroMem supports memory heterogeneity in virtualized environment by making guest-OS heterogeneity aware and enabling effective coordinated hypervisor-guest OS memory management. The contributions are discussed in Chapter 4. Further, to exploit the persistence benefits in memory types such as NVM, this thesis extends beyond memory scaling to provide fast persistent storage in object stores (Chapter 5), and briefly discusses the energy overheads. Finally, the thesis proposes making OS subsystem data structures heterogeneity-aware and achieving scalability in heterogeneity management (Chapter 6).

CHAPTER II

MOTIVATION: IMPACT OF MEMORY HETEROGENEITY

2.1 Applications and heterogeneous memory emulation

An OS-based memory heterogeneity management has to be effective for different class of applications that are memory, compute, storage, and network intensive. Hence, we first study the impact for different classes of applications shown in Table 1 which is in contrast to prior work that have focused mostly in memory intensive applications. The applications analyzed include graph analytics, in-memory data stores, map-reduce computations, as well as database, and web server [20].

Our main goal is to first study the impact of memory heterogeneity on applications by varying the memory bandwidth, latency, and memory capacity, and use the analysis to propose efficient management. The proposed management principles are memory technology independent, hence we emulate

- FastMem: Memory with high bandwidth, low latency, and limited capacity,
- SlowMem: Memory types high access latency low bandwidth, but large capacity. Note that we also study the sensitivity of applications towards different capacity, bandwidth, and latency.

Because only DRAMs are commercially available, to emulate FastMem and SlowMem, we apply DRAM thermal throttling, that can change (reduce) the DRAM latency by up to 5x, and bandwidth by up to 9x. We use memory throttling similar to several prior CPU heterogeneity research that use DVFS because to the best of our knowledge no other mechanisms exist to emulate memory heterogeneity for a full system study. The impact of throttling bandwidth and latency is based on application and OS memory use and does not have other side effects. Figure 2 shows the latency increase, and bandwidth decrease factor with corresponding latency and bandwidth values measured using stream, and memory latency benchmark.

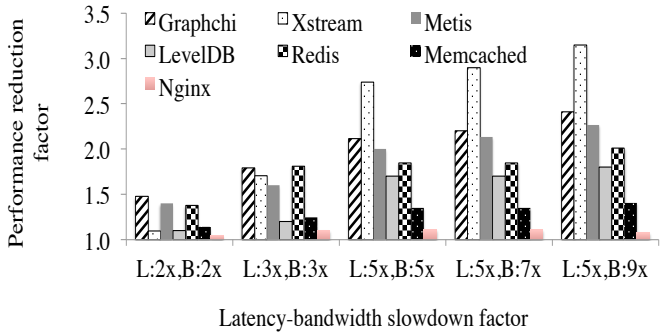
We also verify our analysis using Intel's NVM Emulation Platform (NVMEP) that allows fine-grained emulation of latency and bandwidth using custom BIOS firmware, and special CPU microcode. More details about NVMEP is available in [18]. Because of restricted access to the emulator that inhibits OS-level modification, we use this platform only for verification of our analysis.

Application	Type	Description and Workload	Evaluation Metric
GraphChi [26]	Graph Analytics	Uses Orkut social graph input, 3 million nodes, 117 million edges, requires 8 GB memory [20]	Time(seconds)
XStream [35]	Graph Analytics	Edge-centric graph processing and uses same input as GraphChi	Time(seconds)
Metis [13]	Data Analytics	Shared memory mapreduce that optimizes Phoenix [34], 2GB crime dataset, 8 mapper-reducer threads	Time(seconds)
LevelDB [3]	NoSQL Database	Google's DB for bigtable, SQlite bench with 1M keys	Throughput(MB/s)
Redis [9]	Data caching	Popular in-memory key-value store with support for persistence, Redis benchmark with 2M operations, 80% get	Requests/sec
Memcached [6]	Data caching	In-memory key-value store, 200 sec memslap benchmark run, 80% get	Throughput(MB/s)
NGinx [8]	Web serving	Popular webserver, 1 million static,dynamic, images webpages	Requests/sec

Table 1: Applications

Latency & BW Throttling factor	Latency (nsec)	BW(MB/sec)
L:1x:B:1x	203	11251
L:2x,B:2x	436	5187
L:3x,B:3x	616	3719
L:5x,B:5x	960	2452
L:5x,B:9x	960	1234
L:5x,B:12x	950	907

Figure 2: Bandwidth and latency throttling values.
L:x indicates increase in latency, and B:y indicates reduction in bandwidth.



	Graphchi	Xstream	Metis	LevelDB	Redis	Memcached	Nginx
MPKI	22.4	27.8	9.61	4.7	13.29	16.52	3.21

Figure 3: Latency-bandwidth sensitivity.
In x-axis L:x,B:y represents latency increase factor x, and bandwidth reduction factor y relative to FastMem(L:1x,B:1x). Y-axis represents performance reduction relative to FastMem.

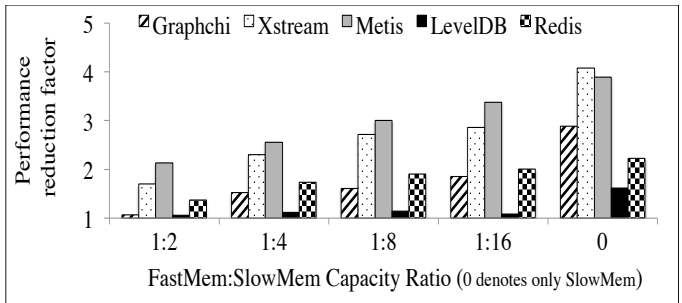
2.2 Impact of memory latency, bandwidth, and capacity

We first study the impact of memory latency and bandwidth followed by sensitivity towards FastMem capacity.

2.2.1 Memory latency, bandwidth sensitivity

In Figure 3, the x-axis shows the memory latency increase and bandwidth reduction factors, and y-axis shows the corresponding performance reduction factor. The table in Figure 3 shows the memory intensity of applications in terms of misses per million instructions (MPKI).
Observations. As expected, application with higher MPKI have a high impact from higher latency, and lower bandwidth. Interestingly, for most applications except graph analytics (XStream and Graphchi), the impact of memory bandwidth reduction is relatively lower compared to the increase in memory latency. For instance, when the latency is reduced by 5x, the throughput of Redis and Memcached reduces by up to 2x and 1.5x respectively. However, with further reduction of only the bandwidth, but same latency, the impact is relatively small. Similar results can be observed for in-memory MapReduce application Metis. For X-stream and Graphchi, with high random access shown by the MPKI values, both latency and memory bandwidth impact performance.

For I/O intensive LevelDB, the throughput reduces only by 30% when the latency is reduced by 3x, but with 5x, interestingly, the throughput reduces by 68% and remains stable even with bandwidth reduction of 9x. This is because, I/O intensive applications use buffer cache in short burst, and suffer from access latency delay. The memory efficient Nginx Webserver with an overall



Capacity vs. Miss Ratio	Graphchi	Xstream	Metis	LevelDB	Redis
1/2	0.3	0.3	0.6	0.0	0.2
1/4	0.7	0.8	0.7	0.0	0.5
1/8	0.8	0.9	0.8	0.0	0.7

Figure 4: (A) Figure shows impact of FastMem capacity. (B) Table shows allocation miss ratio

memory usage of 400MB-500MB has a minimal impact due to memory performance. Although they are I/O intensive, the buffer cache usage for serving a webpage is only a few kilobytes.
Insights. We learn two important insights. (1) Applications exhibit high variability in their sensitivity towards memory speeds, demonstrating the importance of software/OS controlled memory management that adapts to the workload characteristics. (2) The memory latency and bandwidth not only impacts applications that are memory intensive, but it also impacts storage and network intensive applications. Note that, we also verified all the observations in the Intel NVMEP emulator where bandwidth and latency can be modified independently, and found similar trends. The results are available in the Chapter 7.

2.2.2 How sensitive are applications towards FastMem capacity?

Figure 4 analyzes the impact of FastMem capacity. The y-axis shows the ratio of FastMem to SlowMem capacity. The table in the Figure 4 shows FastMem allocation miss-ratio [39] for the 1/2, 1/4, and 1/8 capacity ratio. The miss ratio is the total FastMem page allocation misses to the actual page allocations for a workload.
Observations. When the capacity ratio is 1/2 most applications except XStream, and Graphchi experiences less than 30% overhead. This indicates that (1) either the actual number of allocated and used pages is less than FastMem capacity, or (2) applications allocate and release memory frequently. Reducing the capacity further to 1/4 increases the application slowdown not only for memory intensive applications such as Metis, but also for the I/O buffer cache-intensive applications such as XStream, and GraphChi [26]. This is because current OS prioritize heap allocations with most I/O pages allocated from slow memory. LevelDB with short transactions and use of buffer cache only for read operations (writes are flushed, synced) shows less that 15% up to 1:16 ratio.
Insights. Large scale applications frequently allocate/release memory. Providing flexible memory placement mechanisms that not only prioritize heap allocations but also buffer cache allocations can increase application performance.

2.3 State of the art heterogeneous memory management

Several recent hardware and software research have discussed support for heterogeneous memory [27, 16, 18, 30, 32, 15, 14] in a single computer system. Proposals such as [27, 14, 30, 32] rely on page migration techniques across fast and slower memory. Proposals such as [16, 18] have proposed memory device specific management methods. Surprisingly, most solutions either do not seamlessly scale across different memory devices or provide a smart memory allocation (placement) mechanism. They completely rely on expensive page migration techniques. We first discuss the device specific solutions followed by the drawbacks of migration-based mechanisms.

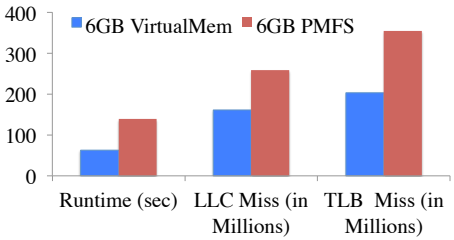


Figure 5: Performance of virtual memory vs. PMFS managed SlowMem in Metis

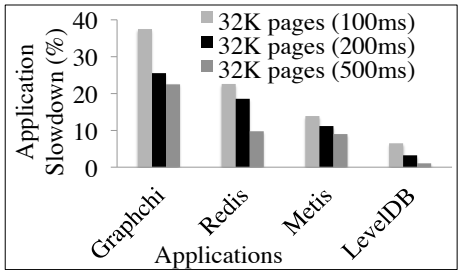


Figure 6: Hot page tracking overhead

2.3.1 Lack of capacity scaling and TLB, Cache inefficiency

Recent research such as PMFS [18] propose a VFS-based management of the SlowMem (NVMs). Note that these solutions not only use SlowMem for storage but also for higher memory capacity (e.g., via mmap with MAP_PRIVATE to the VFS). However, a significant limitation of using a VFS for additional memory capacity is that it cannot seamlessly scale across different memory devices such as FastMem, or DRAM. This is because a VFS can only manage a filesystem, and whereas the virtual memory cannot manage a region managed by the VFS. This incompatibility results in a lack of memory scaling across heterogeneous memory resource. Hence, applications that exhaust one memory type (e.g., FastMem) cannot seamlessly switch SlowMem or vice versa.

Apart from capacity scaling, even when application capacity requirements can be satisfied with VFS-managed SlowMem, using VFS for capacity scaling is highly inefficient of cache and TLB use. This is mainly because of two reasons, (1) VFS data structures were originally designed for block-based disk access and hence processor cache efficiency was not a key consideration, (2) VFS does not classify the persistent use of SlowMem from non-persistent (capacity) use just as for additional capacity. Consequently, for every allocated page several filesystem metadata structures are updated. Note that, in memory-based filesystem such as PMFS, these updates have to be flushed from cache too. Figure 15(c) shows the impact of running Metis on a 6GB SlowMem managed by virtual memory or VFS-based PMFS. Clearly, with the latter, the application runtime, TLB misses, and last-level cache misses overheads are significant. This motivates the need for a VM-based management of all memory types.

2.3.2 Expensive hot page migrations.

Several recent proposals such as [27, 14, 30, 32] have proposed hotness-based page migration techniques across heterogeneous memory types. While this provides seamless scaling, hotness-based page migrations can be significantly expensive. A hotness tracking mechanism has to periodically scan an application page table, disable the reference bit of all the pages in the page table, and for each page accessed subsequently, the access counter of the pages has to be incremented. When a counter reaches a threshold, the pages have to be migrated. First, scanning the entire page table for an application periodically can induce high overhead. Note that failure to scan and migrate in a timely manner would mitigate all the benefits. Next, migrations are significantly expensive not just because of the data movement, but also from the cost of page table update, TLB invalidation, and cache pollution. As evident from Figure 6, even 32K pages in a 100ms interval shows up to 30% overhead, which only decreases by 10% for a 500ms scan interval. Therefore, hotness-based migration techniques are insufficient.

2.4 Lack of memory heterogeneity support in virtualized systems.

The current virtualized system software is designed for homogeneous memory. The hypervisor neither expose memory heterogeneity nor allow flexible use of heterogeneous memory by guest-OS. Hypervisors view the entire guest-OS as an application, with minimal or no information about how applications use memory. Lack of information about applications use of memory limits the hypervisor to use reactive management techniques - monitor hotness of pages and migrate them

accordingly. Lack of coordinated management between the guest-OS and VMM restrict several opportunities such as on-demand allocation to the right memory without the need for migration or using OS semantics for memory placement.

CHAPTER III

APPROACH OVERVIEW AND HETEROMEM OS SUPPORT

3.1 Approach overview

We design HeteroMem (heterogeneous memory), a system software that manages memory heterogeneity by subscribing to the following design principles.

- **Scaling memory by managing heterogeneous memory as NUMA nodes.** HeteroMem treats different memory types as a NUMA node. It then manages each NUMA node by extending the virtual memory, and adding an OS-level specialization for managing them. For example, because NVMs can be used for capacity, and persistence, a specialized NVM manager added as an extension to virtual memory must handle this classification with specialized page allocation. We discuss the details in Section 3.2.
- **Extending virtual memory for scaling, cache, and TLB efficiency.** Unlike prior designs, we extend the OS VM subsystem for managing different heterogeneous memory such as FastMem, SlowMem. Using VM subsystem provides seamless scaling across different memory devices, provides a capability to reuse decades of research and optimizations VM subsystem, and more importantly, inherit the TLB and cache efficient OS data structures.
- **NUMA-based library support for memory placement.** HeteroMem by providing a NUMA-based support provides applications with flexible memory placement policies across heterogeneous memory. It also provides support for extending more such policies.
- **Transparent memory placement by exploiting OS-level information.** HeteroMem exploits the semantic information collected from application's VM use such as page types, their current state such as active and inactive use, and the purpose of page use to make intelligent and application-transparent memory placement across heterogeneous memory.
- **Memory heterogeneity support in virtualized systems.** HeteroMem extends its design to support memory heterogeneity in virtualized systems. Unlike the prior approach of keeping heterogeneity guest-OS transparent, HeteroMem proposes a coordinated heterogeneity management between the hypervisor and the guest-OS for reducing data movement across memory types.
- **Extending virtual memory for persistence in NVMs.** To exploit the persistent properties of NVMs, HeteroMem extends the virtual memory for supporting object-based persistence.

We discuss each of the design principles in detail.

3.2 HeteroMem OS design

We next discuss the HeteroMem design into OS-level (1) hardware abstraction, (2) software abstraction, and finally, (3) OS-level support if the SlowMem supports persistence.

3.2.1 HeteroMem OS hardware abstraction

HeteroMem as NUMA nodes. To support heterogeneous memory allocation and management, HeteroMem leverages existing OS-level abstractions and mechanisms that deal with diverse memory properties, and represents different memories (SlowMem and FastMem) as OS memory nodes. This is similar to what is currently used for memory nodes in NUMA platforms, and also provides flexible support for other future memory technologies. Information regarding available nodes and their

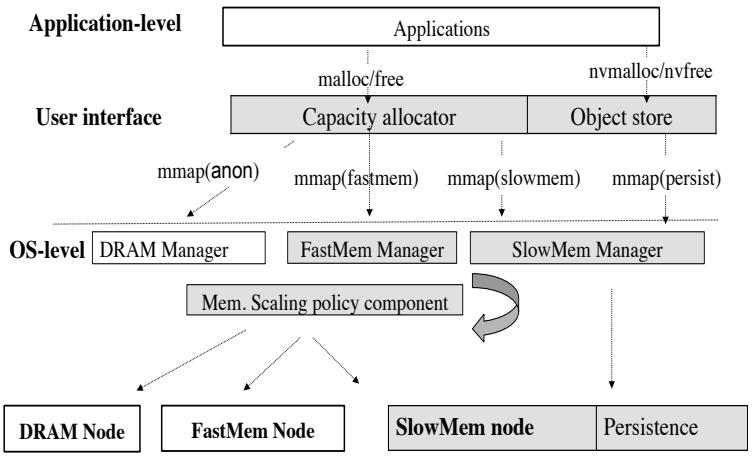


Figure 7: HeteroMem high-level OS view

capacity are exported by the BIOS during initialization. The node-based design supports multiple heterogeneous nodes, with inherent support for features like hot-plugging.

Memory types specific NUMA node properties. In addition to treating different memory types as NUMA nodes, the heterogeneity management also considers the characteristic features of different memory type. For instance, while the FastMem provides high bandwidth low latency memory access, but given the limitations in their capacity, the OS avoids any random allocations to the FastMem, such as different background services, kernel allocations etc. Importantly, HeteroMem disables the default node interleaving NUMA allocation policies used in a typical NUMA system specifically for FastMem. Similarly, for SlowMem such as NVM, HeteroMem's node management is aware of the fact that NVMs can be used for non-persistent capacity as well as persistence use. A common way to segregate these special memory properties is by using of memory zones that partitions a contiguous region of memory based on its usage type. For example in Linux, a DRAM, a node is partitioned into a movable/unmovable, reserved, KERNEL, DMA, HIGHMEM (application allocations) zones for satisfying allocation requests from different subsystems. For FastMem with limited capacity, currently HeteroMem uses only the HIGHMEM, and reserved zones, whereas for NVM, HeteroMem provides also an additional persistence (NVMPERSIST) zone. Extending the zone-level partitioning for different memory types enable HeteroMem to exploit and reuse the capabilities of the highly optimized Linux 'Buddy allocator.

HeteroMem NUMA policies. The purpose of NUMA node-based design is to not only provide a cleaner OS-level abstraction, but also provide flexible memory placement mechanisms for applications, libraries, and OS subsystems. Current state of the art heterogeneous memory management mechanisms (such as PMFS for managing NVMs) lack flexibility to exploit several features of VM subsystem. By extending the virtual memory and NUMA management for different memory technologies, HeteroMem adds different memory placement policies at the OS, and exposes them via. existing user-level NUMA placement libraries. For example, HeteroMem provides three SlowMem specific policies, $SLOWMEM\_REVERT$ that reverts to SlowMem after exhausting FastMem capacity, $SLOWMEM\_PREFER$ which can use SlowMem as a preferred node for applications with higher capacity needs, and lesser or no impact from memory bandwidth and latency. $SLOWMEM\_BIND$ that only uses SlowMem say a background service. Many more policies for different memory types can be easily added with HeteroMem.

3.2.2 HeteroMem OS software abstraction

In addition to the OS-level hardware abstraction of the heterogeneous memory, it is important to classify and partition the OS data structures across different layers of VM. The VM layer includes

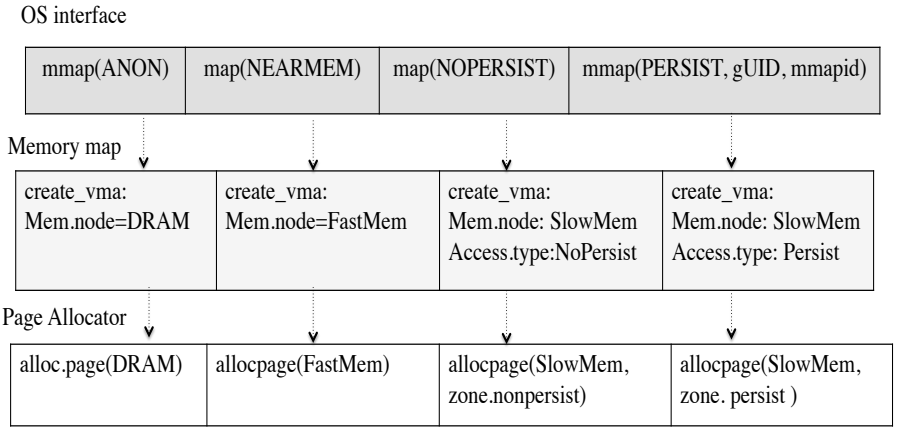


Figure 8: OS map of HeteroMem's virtual memory extension

the memory-access layer that exposes system calls such as mmap to the applications, the memory-mapping layer responsible for creation or update of virtual memory area structures (also referred as VMA), the low-level page allocator that is responsible for allocation of pages from the right memory type as shown in the Figure 3.2. Note that, a VMA refers to a physically contiguous region of memory pages, with all pages in a VMA having same access permission, and used for similar functionality such as additional heap, code pages, etc. We next discuss each layer individually.

Memory access classification. HeteroMem provides applications with explicit and memory-type directed allocation using the existing mmap system call that requires special flags for each memory type. For FastMem, and non-persistent allocations to SlowMem, applications use $MAP\_FASTMEM$, and "MAP_NVMANON" respectively, whereas for persistent allocations, applications use $MAP\_NVMPERSIST$. It is important to note that current VFS-based mechanisms for SlowMem do not differentiate between persistent and non-persistent allocations.

Memory mapping classification. The VMA structure is the heart of VM subsystem, used by the page fault handlers to low-level allocators for mapping a relationship between a page and the memory type and the region to which it belongs to. Hence, HeteroMem extends the memory mapping layer for creating a VMA of specific memory type (SlowMem, FastMem) using the mmap flags, and then also merges VMAs with overlapping memory region provided the VMAs represent same memory type. This is because in the case of SlowMem, mixing persistent and non-persistent memory regions is undesirable. Note that, each process can have several VMAs depending on the capacity usage. The VMA structures are maintained in a per-process VMA tree. Merging contiguous VMAs can reduce the VMA lookup time. Another important scenario to consider is the application transparent allocation of pages, say I/O buffer cache pages. When the OS transparently allocates pages from different memory zones, it creates a VMA of a specific memory type with appropriate flags before adding a page to it.

Page allocation and free CPU list. Generally, the low-level OS page allocator uses the VMA type to identify its corresponding zone and node for a page allocation request. Linux uses 'Buddy' allocator that strives to reduce memory fragmentation caused by non-contiguous allocations by user-level, driver, and kernel subsystems. However, the allocator logic is complex – often called as a slow path, and can dramatically increase the time spent on page allocation. Hence, for fast path allocation, the memory manager uses a per-CPU free page list which is a cache of recently released pages that can be reused. However, current VM supports only homogeneous memory (DRAM). HeteroMem extends the per-CPU list with a nper-CPU list array of different memory types and zones such as FastMem, and SlowMem. Note that, for SlowMem, the zones include persistent and non-persistent zones. We will discuss the OS-level persistence management shortly.

3.3 HeteroMem OS-level memory placement

We next discuss HeteroMems OS-level mechanisms for application transparent memory placement. The mechanisms use OS-level virtual memory information to reason about how an application uses memory by using the region-, page-level information, and the page state values. Using these details we provide principles for allocating it right even before any migrations. Next, given that the memory types vary significantly in their capacity, a right allocation is not always possible. Hence, we focus on mechanisms that efficiently migrate data/pages across memory devices.

Prioritizing heap and I/O cache allocations to FastMem Large scale applications spend a significant time accessing heap memory. More importantly, applications frequently allocate and release memory in their life cycle. Therefore, prioritizing heap page allocations to FastMem can significantly speed up memory intensive applications. Further, frequent page allocation and release cycle increase page reuse, reduces FastMem page allocation misses, resulting in fewer migrations between FastMem and SlowMem. For example, in applications such as GraphChi – a graph computation engine can achieve around 90% of the maximum speed up even when the capacity of FastMem is half the size of DRAM. Further, the time spent on page allocation for FastMem is further reduced with FastMem per-CPU page free-list discussed earlier.

While prioritizing heap allocations to FastMem is beneficial for memory-intensive applications, several datacenter applications such as databases, graph analytics [26, 35, 3] are also I/O (storage and network) intensive. For such applications, the OS buffer cache is highly useful in improving the I/O bandwidth and network performance. It is important to note that the buffer cache pages are short-lived - active for a short duration during I/O access, and inactive or released when free available memory reaches a swap threshold [25]. Hence, the FastMem pages can be reused at different phases of an application. Allocating I/O buffer cache pages on-demand from FastMem improves application speedup for I/O intensive applications. We shortly discuss methods to reduce heap and I/O allocation contention.

Efficient migration with OS-hints and reducing the migration scope Proactive allocations are only possible when the active memory footprint of the application fits within FastMem. When the footprint exceeds the available FastMem capacity, the OS has to resort to page migrations by migrating hot pages from SlowMem to FastMem and vice versa. To avoid the cost of hotness tracking, we propose the following method. The OS already uses LRU-based algorithms for classifying active and inactive referenced pages as a part of OS swap management. Although the classification is not done frequently to reduce additional overhead, however, even before any hotness tracking is enabled, this information can be used to evict all inactive pages. Note that unlike the DRAM-based swap management, limited capacity of FastMem requires an aggressive eviction of inactive FastMem pages to SlowMem. Exploiting the existing OS classification is useful in reducing the short-lived I/O buffer cache pages contention, and increases FastMem reuse. Further, evictions and migration done in batches can significantly reduce the page walk overheads.

To further reduce the scope of hotness tracking and migration, HeteroMem restricts the hotness monitoring only to contiguous application-level heap memory regions (VMAs mapped to heap region)of a process [21] , and avoids hotness tracking for all the linearly-mapped ZONE_DMA pages, and the I/O buffer cache pages, which are anyway evicted by the HeteroMems FastMem specific LRU mechanism. These optimizations significantly reduce the scope of monitoring.

Architectural hints to reduce migration overhead Hardware-level hotness tracking [30] mechanisms that can capture both page reuse and access pattern are unavailable. However, S/W mechanisms can only provide information about page reuse by forcing the OS to count page reference, but such techniques do not provide memory access pattern. Access pattern before migration is important because, applications with high page reuse, but low cache misses cannot benefit significantly from page migration. To address this, HeteroMem combines the cache access behavior of application using hardware counters with the reuse information to decide the migration frequency. High

[0]In Linux, this can be extracted from VMA structures

cache misses increases migration, whereas an application with low cache misses is not impacted by FastMem bandwidth or latency (for example, a Web server).

CHAPTER IV

HETEROMEM SUPPORT FOR VIRTUALIZED SYSTEMS

HeteroMem design is generic that has been easily extended to virtualized systems too. HeteroMem makes the guest-OS heterogeneity-aware providing the same specialized management of different memory types (such as SlowMem, FastMem) in a non-virtualized OS. HeteroMem not only benefits from guest-OS information about an application's memory use but also achieves a design consistent with the principle of thin VMMs [19]. We next discuss important design principles.

4.1 Heterogeneity-aware guest-OS and on-demand allocation.

HeteroMem provides virtual machines (referred as guest-OS hereafter) with an on-demand allocation capability from memory types. The term "on-demand" refers to the wide range of allocation such as page-fault on the first touch for an anonymous or file system memory page, or an OS request for an I/O buffer cache page. The memory types are treated as a guest-OS aware NUMA node. Further, for achieving application transparency, OS-level information discussed earlier in prior sections is used to make 'right' allocations the first time, thus reducing the need for migrations. Enabling guest-OS to demand reservation and allocation of a specific type of memory requires additional hypervisor-level interfaces and fall-back policies, because the hypervisor is responsible for memory allocations and page table updates. In the top section of Figure 9, the guest-OS contains a memory-specific manager as discussed in the previous sections, and an on-demand allocation component that forwards the OS and application request of pages from specific memory type to the hypervisor. We add specialized hypercalls to the Xen memory management, that allows guest to specify the amount and type of requested memory.

4.2 Hypervisor-guest support for coordinated management.

The hypervisor has direct control over system hardware resources and provides a holistic view of system resource information, whereas the guest OS has fine-grained information about the application running over it. Hence, delegating all hardware resource tracking functionality to the hypervisor and using the resource tracking information to take actions at the guest OS-level improves the management efficiency. Given its highest access privilege, the hypervisor is delegated with the responsibility to perform the page-level tracking (hotness tracking) to identify frequently accessed pages. The hotness information is exported to the guest OS, and actual migrations are controlled by the guest, which use its information about application for adaptive and scalable migrations. This is important because, with increasing application working set size, and the VMs on a single host, the number of pages for migration also can increase. Not all migrations are useful, and the limited information about current page state of an application can result in high monitoring and movement overhead. However, the guest-OS, with fine-grained information about such application page state, can selectively customize, or enable/disable the migrations. In Figure 9, each hypervisor has an on-demand allocation component, a memory hotness tracking component, and a fair share resource management component. The hypervisor exports the hotness information through the shared memory communication channel discussed next.

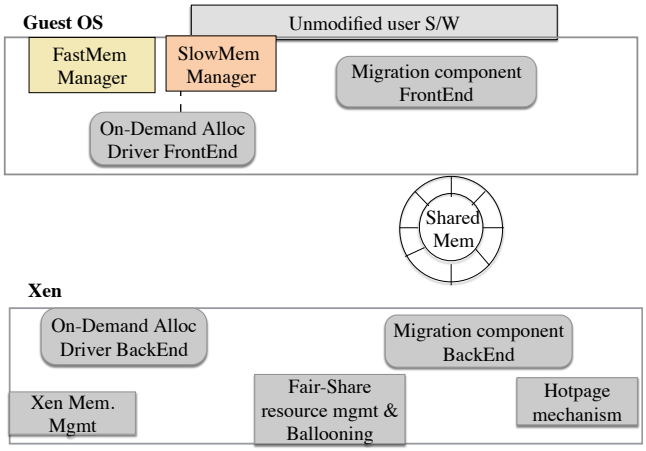


Figure 9: HeteroMem components for virtualized environments. More details are available in the paper.

4.3 Fast communication using shared memory.

Coordinated hypervisor-guest management requires frequent communication (tens of milliseconds) of significantly large data between the guest-OS and the hypervisor. The data includes memory hotness tracking information (hotness), hardware performance counter data, or the hints from the guest-OS to the hypervisor about what and when to scan for page hotness. In virtualized environments, communication happens using a hypercall. The overheads associated with a hypercall can lead to cache and TLB pollution (up to 2/3 of the L1 cache and TLB [36]). This can jeopardize performance. Hence, HeteroMem avoids hypercalls and incorporates a shared memory-based interface among guest OSs and the hypervisor.

4.4 Application transparency.

HeteroMem is an application-transparent approach, and only requires guest-OS- and hypervisor-awareness for memory heterogeneity, thus avoiding the need for new programming models or additional application development complexities. Heterogeneity can be selectively enabled using an application independent shared library to initialize OS HeteroMem support. Further, using a NUMA node-based abstraction provides generic application support for existing NUMA libraries [5].

4.5 Impact of OS-level awareness

To understand the impact of heterogeneity-aware guest OSs, we first discuss the implications of OS-aware management such as on-demand heap (Heap-OD), Heap-IO-OD, FastMem specific eviction and page migrations (Heap-IO-OD-LRU). As discussed earlier, the page allocations, eviction, and data movement are done with information available to guest OS (virtual memory). We use the same set of applications discussed earlier in Table 1.

Figure 10 shows the performance of different applications. The y-axis of each sub-figure indicates the performance metric reported by the application and the x-axis shows the FastMem capacity ratio relative to SlowMem. We compare the performance of (1) SlowMem-only - indicated with dotted line, (2) Heap-OD - heap allocation on-demand, (3) Heap-IO-OD - heap plus I/O cache allocation, (4) Heap-IO-OD-LRU – Heap-IO-OD plus HeteroMem's FastMem specific LRU-based evictions mechanism, and finally (5) FastMem-only - indicated with dark line.

Heap-OD gains: Firstly, as expected most of the optimizations provide significant benefits over the

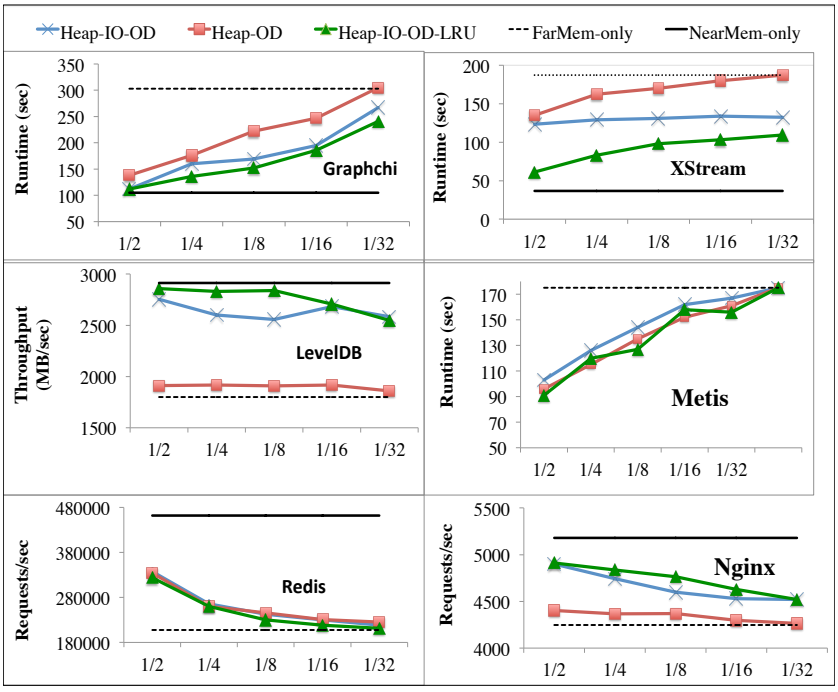


Figure 10: Impact of OS aware design
Heap-OD indicates prioritizing on-demand allocation of heap to the FastMem, Heap-IO-OD indicates Heap + I/O buffer cache allocation, Heap-IO-OD-LRU indicates the proposed LRU-based eviction mechanism.

SlowMem-only approach. Next, even with 1/4th of SlowMem capacity, HeteroMem's Heap-OD approach improves GraphChi, Redis, XStream performance by 2.5x, and by 90% and 52% for LevelDB and Metis. HeteroMem's on-demand FastMem heap allocation provides significant benefits. Next, web server Nginx has limited impact from capacity and B/W. The difference between the FastMem-only and the SlowMem-only approach is 22%. The active heap use is less than 30MB-50MB [7], hence improves performance by around 4%, and ∼60-90MB is used for storage and network buffer which we discuss next.

Impact of Heap-IO-OD: Next, for I/O buffer cache-intensive applications, combining heap and I/O-based on-demand allocations improves speedup over Heap-OD. Even with 1/8 capacity ratio, Heap-IO-OD improves GraphChi, and XStream performance by 23%, and 24% respectively.For XStream, the I/O files are mapped to memory and used for in-place execution also. For the transactional LevelDB, only writes are synced and flushed, whereas reads are mapped to memory, resulting in 28% and 36% improvements for Heap-IO-OD over Heap-OD. For Nginx, by place the network (and filesystem) buffers in FastMem, the gains improved from 2% with Heap-OD to 11.4% with Heap-IO-OD. Finally, an in-memory application like Redis and memory-intensive Metis show no gains. For Metis, the inactive but unreclaimed I/O pages from input causes contention (∼ 6% slowdown) with Heap-OD, but is addressed by Heap-IO-OD-LRU discussed next.

Heap-IO-OD-LRU: This mechanism finds inactive I/O and heap pages marked by the OS, migrates them to SlowMemory, thereby increasing free FastMem pages for subsequent allocations. At 1/4 and 1/8 FastMem capacity for GraphChi, Heap-IO-OD-LRU provides ∼12% improvement over Heap-IO-OD and 32% over Heap-OD. For larger (1/2) or smaller (1/32) capacity, Heap-IO-OD-LRU provides no or minimal benefits, either because of low miss ratio (at 1/2 capacity), or the active working set is much larger than FastMem capacity (at 1/16 and 1/32). We observed 16% improvement for LevelDB, specifically for read operations. For Metis, by evicting inactive pages,

Heap-IO-OD-LRU addresses drawbacks of Heap-IO-OD, but the gains are limited$\sim 3-4\%$.

CHAPTER V

BEYOND CAPACITY: SUPPORT FOR PERSISTENCE IN NVMS

Future SlowMem memory technologies such as byte addressable NVMs can not only support additional capacity but also persistence. The state of the art NVM software and OS solutions [18, 38, 16] have been primarily designed for providing fast persistence by extending the virtual file system (VFS). HeteroMem takes an alternative approach of using the virtual memory based design to provide object-based persistence. Applications map to persistent regions of SlowMem using a user library that provides object store interface and manages the objects stored to the persistent region. Note that HeteroMem's extension are useful for applications such photo stores, IMAP mail server, or key-value stores that use object storage that do not depend on a hierarchical filesystem. We next discuss the motivation of proposing a virtual memory-based persistence, followed by the design.

5.1 NVM filesystem limitations, and Object storage challenges

5.1.1 NVM filesystem S/W overheads.

Prior NVM research such as [37, 18, 16, 38] have extensively studied the overheads of the filesystem even when using NVMs which include frequent kernel mediation required from I/O system calls, metadata update overheads (superblock, inode bitmap, inodes, and other data structures should be updated before a block is modified), metadata concurrency issues from locking, and finally namespace management cost. Recent industry proposals such as [18] have redesigned the OS filesystem metadata to track pages instead of blocks, removing the buffer cache, and providing cache-line size atomic updates. Other research, such as Aerie [37] has proposed a decentralized approach separating trusted filesystem operations (e.g., permission check, metadata update, and integrity), and untrusted operation (e.g., dentry cache, inode namespace management) in the OS vs. user-level, respectively, thereby avoiding frequent kernel mediation and cache pollution.

However, these optimizations do not completely eliminate the software overheads for frequent operations on small files. To analyze an I/O intensive application we use a file compression service – Snappy [11] as a case study in Figure 11(a). Snappy compresses around 2GB of images, videos, emails, and document files. The input size shows high variation in file sizes (kilobytes to gigabytes), requires frequent kernel mediation in the form of system calls, as for every input file, a new output file should be created, and stresses the filesystem metadata updates. Figure 11(a) shows that close

Filesystem time: 60.5%	
Metadata Lock	5.16%
Metadata update	4.46%
copy_from_user	9.58%
copy_user_nocache	9.63%
Page fault	6.11%
Allocation	25.63%

(a) Time spent (%) in PMFS filesystem for Snappy.

```
/*persistent allocation*/
Image**imgdb = nvmalloc("img_root", size);
Start_trans()
  for each new image:
    Image *imgdb[cnt]= nvmalloc(size, NULL);
     cnt++; ……
    /*Commit by flushing and logging
    nvcommit("img_root", size, log=true);
End_trans()
 /* persistent read, implicit load of all child ptrs*/
   img = nvread ("img_root",&size);

 /* non persistent NVM memory allocation */
   tmp = npmalloc(size);
```

(b) Persistent object store programming model.

Figure 11: State of the art NVM filesystem overhead and object store programming.

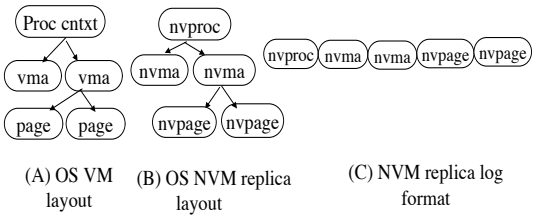


Figure 12: HeteroMem OS persistent structure layout

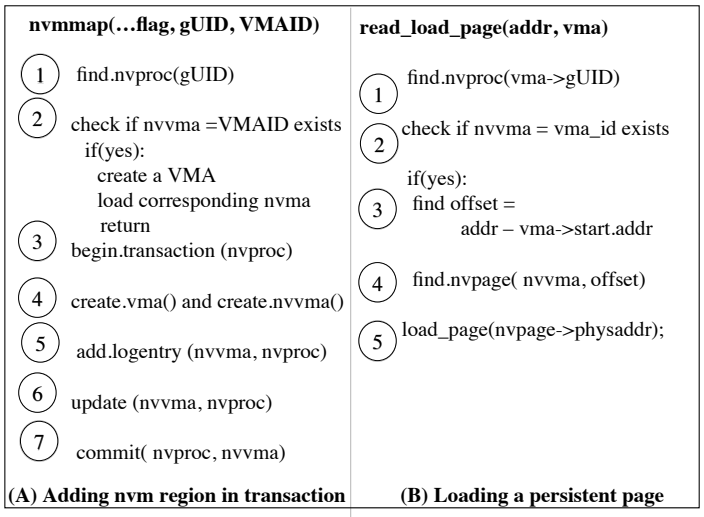


Figure 13: HeteroMem persistent region creation, and page load

to 60% time spent in the OS filesystem. The table also shows a breakdown of the cost across the filesystem components, with significant portions attributed to metadata updates and locks, kernel-to-user buffer data copying, and kernel mediation. updates and locks,

5.1.2 Object storage challenges.

Existing Linux object filesystem such as Ceph [2] and S3fuse [10] provide applications with an object-based put/get interface at the user-level, but map objects to files internally at the filesystem level. Hence, frequent operations on small files/objects suffer from filesystem overheads. To avoid this, prior work [37, 16, 38, 23] have proposed nonvolatile heap objects that reside in some larger mapped (via. mmap) persistent region, and can be accessed/modified with load-store interface. Figure 11(b) shows the programming model for allocating, modifying, committing a simple persistent image object using a persistent heap-based NVM library [23]. Although this avoids direct filesystem interaction/mediation, the mapped regions encapsulating objects are still managed by the VFS. As discussed earlier, VFS suffers from high page access latency, cache, and TLB inefficiency. Another minor drawback of existing persistent object stores is that all persistent objects of an application are mapped to a single large region. As a result, to retrieve even one object, the VFS has to load the entire mapped region increasing the retrieval time. Extending the VM subsystem instead of the VFS for persistent object stores that do not depend on the hierarchical filesystem can reduce page access latency and improve TLB and cache efficiency, and additionally, provide a direct mapping between a persistent object and its related pages, thereby reducing the access time.

5.2 HeteroMem OS persistence support.

Providing support for persistent object stores requires OS and user-level library support. The OS is responsible for persisting all the data structures such as process context, VMA, and pages that form a mapped region, whereas the persistence of the actual content, i.e., application data, is the responsibility of a user-level library. In other words, the OS provides metadata persistence for mapped regions, whereas, the user-library manages/uses the region to provide applications with a persistent object store. We first discuss the OS support for persistent mapped region, and discuss the user-level object store support in the following section.

Complexity of persisting VM structures. When using the VFS, the filesystem, and its metadata already supports persistent mmap regions. In HeteroMem's VM-based design, three important OS structures that form a mapped region require persistence: the process context, all the VMAs inside an application, and finally all page structures inside each VMA. However, persisting these in-memory data structures is significantly complex. This is because, each of these structures dereference other complex OS structures, they are continuously updated by different OS subsystems, and importantly, these data structures are designed for volatile DRAM, and hence are not persistence friendly.

Key Idea: Replica log structures In HeteroMem, we propose a log-based approach which creates a simplified replica of the required process context, VMA, and page structures. The replicas nvproc, nvma, nvpage corresponding to their in-memory process context, VMA, and page structures respectively have only the important information required to locate and load all persistent pages of a region. Figure 12.(A), shows the in-memory state of a process context with VMAs and related pages. Figure 12.(B) in the middle shows the corresponding NVM replica structures (nvproc, nvma, nvpage) in a tree representation.

Figure 12.(C) represents the persistent storage log format of the replica structures. Each process context replica, nvproc, can have a tree of nvma, and each nvma can have a tree of nvpage. Each nvproc is identified by a globally unique ID (gUID) supplied by the user library that maps a region of NVM using the OS interface nvmmap. Each nvma contains a start and end physical address corresponding to the virtual address range of the VMA, and a locally unique VMAID supplied by the user-library (generally an incremental mmap counter). The VMAID is used as an index into the process "nvma" tree. Similarly, the nvpage has a corresponding physical address of the page and an offset from the starting physical address of its VMA. We next discuss the creation and reload/recovery of a persistent mapped region.

Creating and reloading persistent mapped regions . A persistent mapped region is created by an application or an object store library using the OS nvmmap system call. The library also provides a persistent map flag and a globally unique ID (gUID) for each application. HeteroMem's OS-level persistent manager uses the gUID to locate the persistent state of the replica process context in the log, and if not found, creates a new context. Next, a VMA with appropriate persistent flags are created along with their nvma, and added to the replica nvproc's nvma tree, as shown in Figure 12.B. For new persistent pages allocated and added to the in-memory VMA structure, a corresponding nvpage is created and added to the corresponding nvma RB-tree, indexed by the page offset. Figure 13.A,B provides the high-level code for creating a new region, and reading/reloading a persistent region.

To reload/read a persistent region, the application/user-library provides the gUID and the persistent region VMAID maintained in its user-level persistent metadata (explained below). The gUID and VMAID are used to locate the state of the corresponding region in the persistent replica log and load them into a tree structure. The pages are loaded to application address space only after an application touches/access a persistent page in the mapped persistent region. During the first touch, a page fault is generated, and by finding the offset of the faulting address with the starting address of a VMA, a corresponding nvpage structure is identified which contains the physical address of the page to be loaded/added to applications page table. We favor this lazy approach, as opposed to

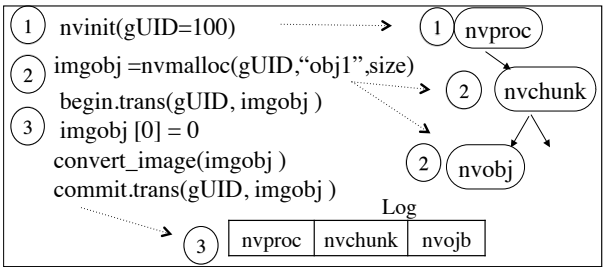


Figure 14: HeteroMem object creation & updates

one which loads all pages at once, in order to reduce restart or read time, and to limit TLB pollution.

Consistency and durability . The OS-layer is only responsible for maintaining consistency and durability of the mapped persistent region, whereas the user-level object store handles it for the actual application content. HeteroMem provides highly optimized OS-level UNDO journaling with atomic commits for cache-level updates for the replica structures. It maintains a separate journal for each gUID or process replica structures, with a global master journal bookkeeping the location of individual process replica journal. Having multiple journals, specifically for frequent page updates avoids contention for a single journal lock across applications. Concurrent updates to a persistent object should be handled by the application/user library. Details of the replica journaling can be found in the attached paper title pVM.

5.2.1 HeteroMem-lib allocator and object store

We next discuss HeteroMems user-level library that provides an application with transparent NVM allocator for capacity use, and object store support.

HeteroMem capacity allocator and NUMA policies. Modern allocators map large regions of memory from the OS and manage them for subsequent allocations. For HeteroMem, we extend the scalable jemalloc [1] library allocator with nvmmap support for directing allocations to NVM. The jemalloc library allocator is used for both non-persistent capacity and persistent allocation with appropriate flags to classify them. By using a library allocator, no application-level changes are required for non-persistent applications. Further, applications can also use the NVM with NUMA policies.

HeteroMem object store interface HeteroMem's object store interface and ACID mechanisms are inspired from prior NVM-as-heap research [16, 38] to create named objects. We first briefly describe the application interface followed by its object store metadata management customized for VM-based OS design. The object store provides applications with a 'nvmalloc' interface to create named, uniquely identified object (objID) formed by combining the per-application UID and the unique object name. Figure 14 shows a sample code to allocate an image object, and use the reference of the allocated object to update the object via a byte addressable load-store interface. To provide consistency and durability, HeteroMem uses the Intel's persistent memory library [23], to wrap updates to an object inside transactions. We discuss the details next.

Persistent object store metadata. An object store creating persistent objects should also maintain persistent metadata about the objects such as their location in mapped regions, size, objID, and information about consistency. Because modern allocators are complex, and maintaining the entire state of allocators in NVM can be expensive, we create a simplified replica structure in a log, similar to that for OS-level persistence. During application initialization, a unique gUID is generated, and a corresponding user-space "nvproc" structure is created in a user-level persistent log as shown in ① in Figure 14. Next, the allocator creates a large persistent region with gUID, and an incrementing MMAPID (used as VMAID by the OS). Upon successful creation of an NVM

region, a corresponding "nvchunk" structure is added to the NVM metadata as in ②. Because each application can have several mmap'ed regions, each nvproc can have several nvchunks. Next, for every object mapped to a persistent region, a corresponding nvobject is added to the nvchunk. To provide consistency and durability, the objects and object store metadata are updated within transactions, using a UNDO log ③. Note that the UNDO log is a journal located separately from the persistent metadata. Our consistency and durability mechanism is borrowed from object-based logs of prior work [16]. For reading/recovering a persistent object, application uses the nvread() interface, with the gUID and object name. The object ID generated from the combination of gUID and object name is used to locate the corresponding persistent region and instruct the HeteroMem's OS manager to load the region to application address space.

5.3 Energy overheads

Next generation byte addressable nonvolatile memory (NVM) like PCM offers memory scalability as well as fast persistent storage. In such environments, however, NVM's limitations of slow writes and high write energy are magnified for applications that require atomic, consistent, isolated and durable (ACID) updates. Specifically, NVM write/store operations and energy consumption are increased because, for satisfying correctness (ACI), persistent application states must be frequently flushed from intermediate buffers like the processor cache, and to support the durability (D), that state must be logged. This significantly increases NVM access energy cost, and more importantly, it results in additional CPU instructions and energy.

To address the energy overheads of persistence, we develop novel energy-aware persistence (EAP) principles that (1) identify data durability (logging) as the dominant factor in energy increase. Next, we (2) formulate energy-efficient durability techniques that reduce those costs. Examples include flexible logging that switches between performance and energy efficient modes, and a memory management technique that trades capacity for energy. Finally, we propose (3) a relaxed durability (ACI-RD) mechanism for critically low energy conditions, that does not affect application correctness. (4) We evaluate the energy benefits and performance implications of using the EAP principles with realistic applications and benchmarks. Our experimental results demonstrate an up to 2x reduction in CPU and NVM energy usage compared to traditional ACID-based persistence. More details about these proposed techniques can be found in the paper attached "Energy Aware Persistence: Reducing Energy Overheads of Persistent Memory via Efficient and Relaxed Durability".

CHAPTER VI

CURRENT STATUS, PROPOSED WORK, AND TIMELINE

6.1 Current status

HeteroMem covers a broad area of memory heterogeneity management across virtualized and non-virtualized systems for different memory types. The overall research discussed in this document has evolved from three different papers that are either ready for submission, or under submission. The papers are attached to this proposal document.

pVM: This paper titled pVM - Persistent virtual memory for efficient capacity scaling and object storage explores the idea of extending OS-virtual memory for supporting memory device such as NVM for not only additional capacity but also providing fast persistent object storage. The paper is under submission to Eurosys 2016.

HeteroMem: This paper discusses the importance of providing a unified support for different memory technologies with a focus towards virtualized system. The proposed system makes the virtual machine heterogeneity-aware and proposes a coordinated management of memory heterogeneity between the hypervisor and the guest-OS. More discussions and related results can be found in the attached paper titled HeteroMem: Coordinated management of heterogeneous memory in virtualized systems.

Energy aware persistence: This research deals with reducing the ACID-based energy overheads when NVMs are used for memory-based persistence. The paper proposes an energy-aware software durability mechanisms that trade-off performance and capacity for reducing energy cost. The paper titled "EAP- Energy aware persistence" is ready for submission and is also attached with this document".

6.2 Proposed work

Previous discussions about HeteroMem proposes a unified abstraction of different memory technologies for virtualized and non-virtualized systems and OS-level extension to virtual memory. Further, HeteroMem also proposes different data placement mechanisms that strictly focus on application's memory, or places the entire subsystem such as buffer cache or network buffers to the fast memory. But however, several improvements are required to make the system software more effective. We next discuss such improvements as the proposed work.

- **Impact on latency and bandwidth bottleneck** While our prior work proposed several OS-level data placement mechanisms that aim to reduce the cost of data movement, we are yet to characterize the effectiveness of such mechanisms in reducing the latency and bandwidth bottlenecks in heterogeneous memory environment.
- **Scaling heterogeneity management across multiple applications** As recent datacenter trends show, the number of processors and memory in a single system is increasing. Consequently, for reducing datacenter equipment, maintenance, and energy consumption, more and more applications or VMs are consolidated into a single large system. Obviously, this is bound to increase the complexity of software in heterogeneous memory environment. However, to satisfy SLAs of different applications, an efficient management that scales up across multiple applications or VMs is necessary. We will propose efficient mechanisms and algorithms for managing memory heterogeneity across multiple applications and VMs.

- **Heterogeneity aware OS subsystem** Most of our prior discussed techniques consider a coarse-grained memory placement, (i.e.,) they place the entire application heap, entire network buffers, or the buffer-cache of filesystems into FastMem. These techniques do not consider the fact that each of the subsystems comprises of several data structures with high variations in memory allocation and access pattern. Given the limited capacity of FastMem, placing the whole subsystem in the FastMem can starve other memory sensitive application/OS data, and data structures. For instance, a network data structure such as 'sk_buff is used across different network layers, and also embeds the metadata, and data for different network layers. Similarly, a filesystem comprises of several metadata and data structures such as dentry caches, filesystem metadata such as superblock, inodes, apart from the actual blocks and buffer caches. But not all the metadata structures have the same access pattern or sensitivity towards memory bandwidth and latency. To the best of our knowledge, no heterogeneous memory placement mechanisms, principles, or literature exists for the efficient subsystem-level placement in heterogeneous memory.

  Hence, the proposed work aims to achieve the following goals, We will analyze the memory placement of commonly used subsystems such as, (1) storage, (2) network, (3) the virtual memory subsystem which includes analysis of the software page table, TLB placement, kernel allocation buffers, and additionally, (4) different concurrency data structures such as locks. The outcome of analysis will propose a generic OS-level algorithm that provides heuristics for prioritizing applications that uses different subsystems.

- **Applications and Evaluations** We will focus on the same set of applications discussed earlier, plus some additional applications that have varying performance behavior with respect to memory allocation patterns, and usage of different subsystems.

The presentation will include more details about the proposed work.

6.3 Timeline

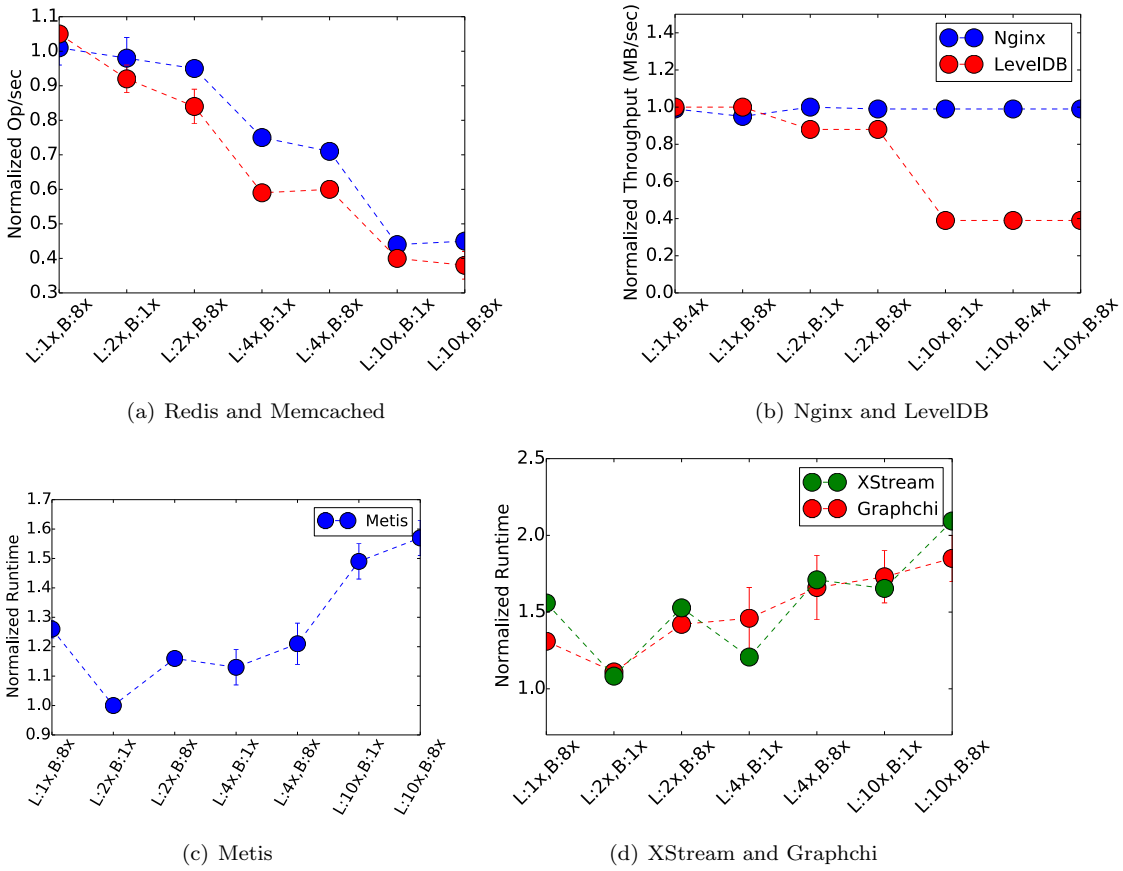
We plan to complete the proposed work by May 2016 for a possible OSDI 2016 submission, or earlier.

CHAPTER VII

APPENDIX

7.1 NVM Emulator platform evaluation

The Figure 15 below shows the impact of latency and bandwidth on applications measured using the NVMEP platform discussed in Chapter 2. As evident, the impact of latency increase is comparatively higher compared to the bandwidth reduction for most applications. The overall impact on applications from bandwidth reduction and latency increase is relatively lower for the same workloads (see Table 1) analyzed in Chapter 2. This is because the NVMEP platform has a larger cache (32MB), higher processor frequency, and may be larger memory buffers.



(a) Redis and Memcached

(b) Nginx and LevelDB

(c) Metis

(d) XStream and Graphchi

Figure 15: NVMEP Emulator: Latency-bandwidth sensitivity of applications. In x-axis L:x,B:y represents latency increase factor x, and bandwidth reduction factor y. Y-axis values normalized relative to FastMem(L:1x,B:1x).

REFERENCES

[1] http://www.canonware.com/jemalloc.

[2] "CEPH file-systems." https://ceph.com/ceph-storage/file-system/.

[3] "Google LevelDB ." http://tinyurl.com/osqd7c8.

[4] "Hybrid Memory Cube Consortia." http://www.hybridmemorycube.org/technology.html.

[5] "Linux libnuma." http://linux.die.net/man/3/numa.

[6] "Memcached." http://memcached.org/.

[7] "Nginx memory usage." https://www.nginx.com/blog/nginx-websockets-performance/.

[8] "NGinx Webserver." http://nginx.org.

[9] "Redis." http://redis.io/.

[10] "S3fuse filesystem." https://github.com/s3fs-fuse/s3fs-fuse.

[11] "Snappy Compession." http://tinyurl.com/ku899co.

[12] Akin, B., Franchetti, F., and Hoe, J. C., "Data reorganization in memory using 3d-stacked dram," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 131–143, ACM, 2015.

[13] Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z., "Corey: An operating system for many cores," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, (Berkeley, CA, USA), pp. 43–57, USENIX Association, 2008.

[14] Chou, C.-C., Jaleel, A., and Qureshi, M., "Batman: Maximizing bandwidth utilization for hybrid memory systems," in *Technical Report, TR-CARET-2015-01 (March 9, 2015)*, 2015.

[15] Chou, C., Jaleel, A., and Qureshi, M. K., "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, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2014.

[16] Coburn, J., Caulfield, A. M., and others, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," in *ASPLOS '11*.

[17] Dong, X., Xie, Y., Muralimanohar, N., and Jouppi, N. P., "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, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.

[18] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J., "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, (New York, NY, USA), pp. 15:1–15:15, ACM, 2014.

[19] Engler, D. R., Kaashoek, M. F., and O'Toole, Jr., J., "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, (New York, NY, USA), pp. 251–266, ACM, 1995.

[20] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B., "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, (New York, NY, USA), pp. 37–48, ACM, 2012.

[21] Gorman, M., "Understanding the Linux Virtual Memory Manager." `http://bit.ly/1n1xIhg`.

[22] Gupta, V., Lee, M., and Schwan, K., "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, (New York, NY, USA), pp. 79–92, ACM, 2015.

[23] Intel, "Logging library." `https://github.com/pmem/nvml`.

[24] Jiang, X., Madan, N., Zhao, L., Upton, M., Iyer, R., Makineni, S., Newell, D., Solihin, D., and Balasubramonian, R., "Chop: Adaptive filter-based dram caching for cmp server platforms," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pp. 1–12, Jan 2010.

[25] Jonathan, C., "Linux Swapping." `https://lwn.net/Articles/495543/`.

[26] Kyrola, A., Blelloch, G., and Guestrin, C., "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 31–46, USENIX Association, 2012.

[27] Lee, B. C., Ipek, E., Mutlu, O., and Burger, D., "Architecting phase change memory as a scalable dram alternative," in *ISCA*, ACM, 2009.

[28] Lee, B. C., Ipek, E., Mutlu, O., and others, "Architecting phase change memory as a scalable dram alternative," in *ISCA '09*.

[29] Loh, G. and Hill, M., "Supporting very large dram caches with compound-access scheduling and missmap," *Micro, IEEE*, vol. 32, pp. 70–78, May 2012.

[30] Meswani, M., Blagodurov, S., Roberts, D., Slice, J., Ignatowski, M., and Loh, G., "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*, pp. 126–136, Feb 2015.

[31] Qureshi, M. K. and Loh, G. H., "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, (Washington, DC, USA), pp. 235–246, IEEE Computer Society, 2012.

[32] Qureshi, M. K., Srinivasan, V., and Rivers, J. A., "Scalable high performance main memory system using phase-change memory technology," in *ISCA '09*.

[33] Ramos, L. E., Gorbatov, E., and Bianchini, R., "Page placement in hybrid memory systems," in *Proceedings of the International Conference on Supercomputing*, ICS '11, (New York, NY, USA), pp. 85–95, ACM, 2011.

[34] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C., "Evaluating mapreduce for multi-core and multiprocessor systems," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 13–24, Feb 2007.

[35] Roy, A., Mihailovic, I., and Zwaenepoel, W., "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 472–488, ACM, 2013.

[36] Soares, L. and Stumm, M., "Flexsc: Flexible system call scheduling with exception-less system calls," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–8, USENIX Association, 2010.

[37] Volos, H., Nalli, S., Panneerselvam, S., Varadarajan, V., Saxena, P., and Swift, M. M., "Aerie: Flexible file-system interfaces to storage-class memory," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, (New York, NY, USA), pp. 14:1–14:14, ACM, 2014.

[38] Volos, H., Tack, A. J., and Swift, M. M., "Mnemosyne: lightweight persistent memory," in *ASPLOS '11*.

[39] Waldspurger, C. A., Park, N., Garthwaite, A., and Ahmad, I., "Efficient mrc construction with shards," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, (Berkeley, CA, USA), pp. 95–110, USENIX Association, 2015.

[40] Zhao, L., Iyer, R., Illikkal, R., and Newell, D., "Exploring dram cache architectures for cmp server platforms," in *Computer Design, 2007. ICCD 2007. 25th International Conference on*, pp. 55–62, Oct 2007.

pVM - Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage

Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan

Abstract

Next generation byte addressable nonvolatile memories (NVMs), such as phase change memory (PCM) and Memristors, promise fast data storage, and importantly, address DRAM scalability issues. State of the art NVM OS mechanisms has focused on improving the block-based virtual file system (VFS) to manage both persistence and capacity (memory scaling) needs of applications. However, using VFS for capacity scaling has several limitations, such as lack of automatic memory scaling across DRAM and NVM, inefficient use of processor cache, TLB, and high page access cost, thereby reducing the application performance. These limitations also impact applications that use NVM for persistent object storage with flat namespace, such as photo store, NoSQL database, and others.

To address such limitations, we propose persistent virtual memory (pVM), a system software that provides applications with (1) automatic OS-level memory scaling, (2) flexible memory placement policies across NVM, and finally, (3) fast object storage. To achieve (1) and (2), pVM extends the OS virtual memory (VM) instead of the VFS, and abstracts NVM as another NUMA node with support for NVM-based memory placement mechanisms. pVM inherits the virtues of cache and TLB efficient VM subsystem, and augments it further by classifying persistence vs. nonpersistent capacity use of NVM. pVM achieves fast persistent storage by further extending the VM subsystem with consistent and durable OS-level persistent metadata. Evaluation of pVM with memory capacity intensive applications show 2.5x speedup, up to 80% lower TLB and cache miss compared to VFS-based systems. pVM's object store provides 2x higher throughput, and up to $\sim$4x reduction in the OS time.

1. Introduction

The volume of data generated by servers as well as end-user and sensor-rich client devices is dramatically increasing. This poses a significant challenge to scale existing memory and data storage technologies. At one end of the spectrum, scaling DRAM density without increasing the cost and energy is becoming critical [10], and at the other end, current block-based storage devices such as NAND flash suffer from significantly lower bandwidth. The widening gap between storage (persistent) vs. volatile memory access is driving researchers to look beyond NAND for alternative nonvolatile memory (NVM) technologies such as PCM [31, 35] and 3DXPoint [1]. Emerging NVMs promise 100x faster access than SSDs, 4x-8x higher density compared to DRAM, and provide byte-addressable load-store interface to applications. In addition to fast storage, and despite the known limitations [31] such as slow writes and high write energy, [31], the density and performance properties provide an opportunity to use NVMs as a cost-efficient alternative to DRAM for extending the memory capacity of future platforms.

However, the state of the art NVM software and OS solutions [19, 22, 40] have been primarily designed for providing fast persistence by extending the virtual file system (VFS), with memory capacity scaling via NVM a secondary feature. Current solutions use the memory mapping capability of VFS to provide additional capacity, however they (1) do not support automatic memory scaling, i.e., seamless use of one memory (NVM), when resources of another memory (DRAM) are exhausted, and (2) lacks flexible memory placement mechanisms between DRAM and NVM. More importantly, (3) the VFS-based solutions do not classify between persistent and capacity (nonpersistent) use, thereby adding filesystem metadata bookkeeping overheads even when application use NVM for additional capacity (nonpersistent) use. This coupled with cache and TLB inefficient VFS datastructures, seriously limit the NVM use for additional capacity. (4) Further, using block-based VFS in NVMs for persistent object stores that do not depend on hierarchical filesystem (e.g., photo stores, IMAP mail server), fails to exploit the optimal NVM capabilities due to filesystem metadata, concurrency overheads coupled with limitations such as (3).

To address the limitations of state of the art solutions, this paper, **persistent Virtual Memory (pVM)**, explores an alternative approach of extending the existing virtual memory (VM) subsystem, instead of the VFS, for achieving the dual benefits of efficient capacity scalability, and persistent storage for NVMs. pVM's design is based on the principles that byte-addressable NVMs resemble 'slower' memory placed in parallel to DRAM with the memory bus as a hardware interface, rather than a faster disk. Hence, extending the virtual memory (VM) subsystem is better suited to more efficiently exploit the dual benefits of NVMs for capacity and persistent object storage. pVM is not a replacement for traditional filesystem, and has been specifically designed for use of NVM for scaling memory, and provide persistent object store. To the best of our knowledge, pVM is the first OS-based design that extends the VM subsystem to provide such dual use.

pVM's OS hardware abstraction treats NVM as NUMA node to which application can transparently or explicitly allocate additional heap memory, as well as store persistent objects using user-level memory persistence libraries [11, 40]. The generic NUMA node-based design can support several NVM nodes, and provide support for scaling memory seamlessly via. the OS, or provide application with flexible NVM-specific memory placement policies that can be easily integrated with existing memory placement libraries. Next, unlike the VFS-based approach that maps both persistent and non-persistent allocations to a file, pVM' OS virtual memory framework classifies persistent and non-persistent memory (page) allocation, and manages them independently, without adding any persistent metadata management cost for capacity (heap) allocations. This significantly reduces page access time, cache and TLB misses critical for application using NVMs with higher device access cost compared to DRAM as shown in Table 1.

Next, regarding persistent storage, by extending the VM subsystem with persistence management, pVM avoids the metadata complexity of a block-based filesystem, and the kernel mediation from using a POSIX interface, thereby significantly reducing the time spent in the OS. pVM maps persistent objects to a region of NVM pages that can be retrieved across application sessions. Moreover, similarly as with the memory scalability benefits, the benefits from fast on-demand page allocations, and efficient TLB and cache use improves object storage performance. pVM is designed for objects stores with flat namespace unlike applications with significant dependence on the hierarchical filesystem e.g., SQL databases, user personal data, search engines. Hence we envision co-existence of pVM, and block-based NVM filesystem used in applications and user functionality that require structured data storage partitioning. pVM differs from prior efforts [17, 19, 20, 40, 43] that are optimized for one dimension – either persistence or capacity.

Attributes	DRAM	PCM	NAND
Byte-Addressable	Yes	Yes	No
Density	1x	2x-4x	4x
Read,Write latency	1x (60ns),1x	1x, 2x-5x	400x
Endurance	10^{16}	10^{8}	10^{4}

Table 1: Comparison of NVM technologies [13, 22].

pVM makes the following technical contributions:

- *End-to-end persistent virtual memory design (Section 3)* – We propose persistent virtual memory (pVM), an end-to-end virtual memory-based design that treats NVM as a memory node (similar to NUMA), and extends the OS VM subsystem for memory scaling, and fast and persistent object stores.
- *Memory scaling and placement support (Section 4)* – We demonstrate several benefits from pVM's VM-based design, such as automatic memory scaling, support for flexible NVM data placement policies compatible with existing NUMA-based libraries, fast page access, and improved processor cache and TLB efficiency, compared to a state-of-the art VFS-based design.
- *Fast persistent object storage (Section 4)* – We extend VM datastructures with support for persistence, and thereby provides fast consistent durable object storage. The mechanism significantly reduces in-kernel time and improves storage performance for object storage.
- *Evaluation and performance (Section 5)* – We evaluate pVM with different memory capacity intensive benchmark and applications. Our evaluation shows 2.5x improvement in performance with automatic memory scaling, further reducing cache and TLB misses by up to 84%, and page access cost by up to 60%. Further, object storage using pVM improves overall throughput by up to 52%-2x relative to VFS-based object store, and up to 4x reduction in the OS time.

2. Background and Motivation

We next discuss the background, memory scaling and persistent storage limitations of current solutions and motivate the need for VM-based support.

2.1 Background and state of the art.

Byte addressable NVMs such as PCM [31, 35] and 3DX-Point [1] persistent devices expected to be 100x faster (read-write performance) compared to current SSDs [17, 20, 22]. Table 1 shows the H/W characteristics, and their differences with DRAM and NAND device. These NVM devices have higher density scaling as they can store multiple bits per cell with no refresh power, with known limitations imposed by an endurance of a few million writes per cell. Further, they can be connected addressed with memory bus, with load-store operations propagating via. process cache, which can reduce some of access latency [33]. Different usage models have been explored for capacity vs. persistence use. Early system software and hardware research [31, 35] explored NVMs as a fast cache behind the DRAM hierar-

chy for gaining additional capacity only without considering the storage aspect. Recent S/W and OS solutions such as [17][20][22] have redesigned the block-based filesystem to suit the memory-storage by extending the VFS data structures. Other research, such as [19, 40, 41] have proposed orthogonal solutions that expose applications with nonvolatile heap objects similar to the well-known LRVM [37] work without the use of POSIX semantics and OS filesystem. But all these approaches internally use the VFS-backed memory pages. In this work we consider both capacity and object storage similar to nonvolatile heaps [19, 40, 41], but the key difference is the use of VM subsystem. We use Intel's PMFS[22] as the state of the art due to several important OS-level optimization, and wide acceptance in the NVM research community.

2.2 NVM capacity use issues

OS capacity scaling limitations The benefits of using NVM for additional memory capacity have been explored in the past [19, 31]. However, prior proposals are designed primarily for persistent storage, and rely on the VFS for gaining additional memory capacity, by using an library allocators that maps an NVM file into an application address space (e.g., via mmap with MAP_PRIVATE in Linux). However, the mapped NVM regions are managed by VFS as opposed to VM that manages DRAM page allocations. Hence, for applications which cannot fit in DRAM (or NVM), and when all the memory resource is exhausted, the OS virtual memory views NVM as a filesystem and not memory, and hence does not seamlessly switch to NVM for additional capacity usage. Such approaches suffer from the lack of ability to transparently scale over multiple memory types, and do not have flexible policies in the OS or application for guiding memory placement across hybrid memory, and therefore, impacts application performance. Figure 1.A shows the impact of running well known memory capacity intensive applications with standard malloc allocator on two configurations: (A) 6GB split across 2 DRAM sockets (3GB each) vs. (B) 3GB DRAM- 3GB NVM running with PMFS. The applications are described in Table 2. While in configuration (A), the OS automatically balances memory allocation across the socket, in (B), the OS can only use 3GB DRAM as NVM is viewed as a filesystem. As a result swapping increases, and impact performance due to memory pressure.

Cache, TLB inefficiency of VFS. Apart from capacity scaling, even when application capacity requirements can be satisfied with VFS-managed NVM, using VFS induces significant software overheads in the form of high cache and TLB miss, reducing application performance. This is because current VFS-based mechanisms, do not classify persistent use of NVM from non-persistent use just as for additional capacity. As a result, for every allocated page several filesystem metadata structures are updated, and journaled too. Further, the metadata structures like inodes, superblocks were originally designed for block-based access, hence highly cache

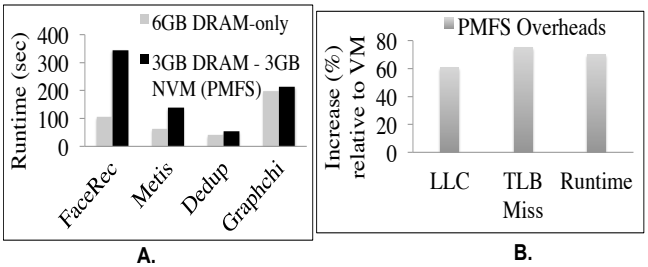


Figure 1: Capacity use.(A) Performance under limited DRAM capacity, (B) Metis S/W overheads when using PMFS exclusively for capacity.

Filesystem time: 60.5%	
Metadata Lock	5.16%
Metadata update	4.46%
copy_from_user	9.58%
copy_user_nocache	9.63%
Page fault	6.11%
Allocation	25.63%

Figure 2: Time spent (%) in filesystem for Snappy app.

and TLB inefficient. Figure 1.B shows the high cache and TLB misses (up to 80% increase) for the Metis application, even when its capacity requirement fits in the NVM memory by reducing the application input size. *These results motivate the need for VM-based design of the NVM OS software stack.*

2.3 Persistent storage limitations

NVM filesystem S/W overheads. Prior NVM research such as [19, 22, 40, 41] have extensively studied the overheads of the filesystem even when using NVMs which include frequent kernel mediation required from I/O system calls, metadata update overheads (superblocks, inode bitmaps, inodes, and other datastructures should be updated before a block is modified), metadata concurrency issues from locking, and finally namespace management cost. Recent industry proposals such as [22] have redesigned the OS filesystem metadata to track pages instead of blocks, removing the buffer caches , and providing cache-line size atomic updates. Other research, such as Aerie [41] has proposed a decentralized approach separating trusted filesystem operations (e.g., permission check, metadata update and integrity), and untrusted operation (e.g., dentry cache, inode namespace management) in the OS vs. user-level, respectively, thereby avoiding frequent kernel mediation and cache pollution.

However, these optimizations do not completely eliminate the software overheads for frequent operations on small files. To analyze an I/O intensive application we use a file compression service – Snappy [6] as a case study in Figure 2. Snappy compresses around 2GB of image, video, email, and document files. The input size shows high variation in file sizes (kilobytes to gigabytes), requires frequent kernel mediation in the form of system calls, as for every input file, a new output file should be created, and stresses the filesystem metadata updates. Figure 2 shows that close to 60% time

```
/*pMem persistent allocation*/
Image**imgdb = nvmalloc("img_root", size);
Start_trans()
  for each new image:
     Image *imgdb[cnt]= nvmalloc(size, NULL);
      cnt++;  ……
    /*Commit by flushing and logging
    nvcommit("img_root", size, log=true);
End_trans()
 /* persistent read, implicit load of all child ptrs*/
   img = nvread ("img_root",&size);

 /* non persistent NVM memory allocation */
   tmp = npmalloc(size);
```

Figure 3: Persistent object store programming model.

spent in the OS filesystem. The table also shows a breakdown of the cost across the filesystem components, with significant portions attributed to metadata updates and locks, kernel-to-user buffer data copying, and kernel mediation. Even using the mmap/munmap interface does not solve the problem for small files as every mmap call is supported by other system calls (open, close, stat), and can aggravate the issue. *These results show that, the OS file-system overhead can be significant even in the state of the art NVM filesystem.*

Object storage challenges. Existing Linux object filesystem such as Ceph [7] and S3fuse [5] provide applications with an object-based put/get interface at the user-level, but map objects to files internally at the filesystem level. Hence, frequent operations on small files/objects suffer from filesystem overheads. To avoid this, prior work [19, 29, 40, 41] have proposed nonvolatile heap objects that reside in some larger mapped (via. mmap) persistent region, and can be accessed/modified with load-store interface. Figure 3 shows the programming model for allocating, modifying, committing a simple persistent image object using an persistent heap-based NVM library [29]. Although this avoids direct filesystem interaction/mediation, the mapped regions encapsulating objects are still managed by the VFS. As discussed earlier, VFS suffers from high page access latency, cache and TLB inefficiency. Another minor drawback of existing persistent object stores is that, all persistent objects of an application are mapped to a single large region. As a result, to retrieve even one object, the VFS has to load the entire mapped region increasing the retrieval time. *Extending the VM subsystem instead of the VFS for persistent object stores that do not depend on hierarchical filesystem can reduce page access latency and improve TLB and cache efficiency, and additionally, provide a direct mapping between a persistent object and its related pages, thereby reducing the access time.* These observations motivate the design and implementation of pVM, described in the following sections.

3. pVM Overview

Setting. pVM is designed for byte addressable NVM technologies such as PCM [9, 34, 35] and 3DXpoint [1] connected via the memory-controller, and placed in parallel with DRAM. NVMs can be accessed by the CPU/applications with load-store instructions to NVM pages via hardware supported page table [12], and exhibit lower access latency and bandwidth compared to DRAM. Since NVMs like this are not available yet, we emulate them using a dedicated NUMA socket, and apply memory thermal throttling to reduce the bandwidth up to 10x, and read vs. write latency by 2x vs. 5x, relative to DRAM. To account for the differing read vs. write latency, we use a software injected delay using LLC load-store miss model used by [20, 22, 39]. Such emulations have to be done even when using currently available NVDIMM (DRAM backed by SSD) which are PCI-E-connected (as opposed to memory controller-based), and with latency/bandwidth close to DRAM. Further, NVDIMMs are limited by DRAM scalability.

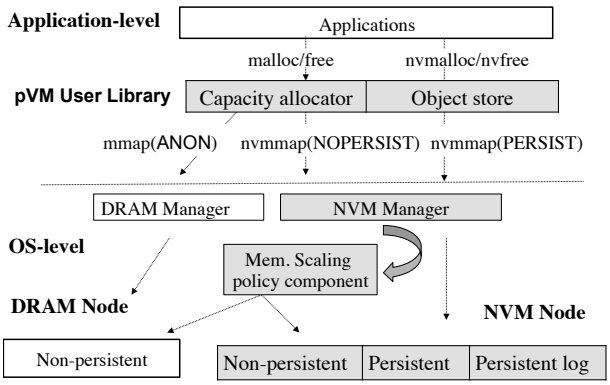


Figure 4: pVM-High Level Design. Shaded blocks indicate pVMs S/W-level stack changes

Approach Overview. We build pVM to provide applications with the following important capabilities: (1) *scalable capacity* – additional memory capacity to achieve application transparent memory scaling similar to multi-socket NUMA machines, (2) *efficient persistence* – fast heap-based persistent object store, for applications not requiring the filesystem hierarchy, and (3) *improved performance* – reduced cache and TLB misses when using NVMs, and improved performance.

Figure 4 illustrates the high-level overview of the pVM design. For scaling capacity, pVM treats NVM as a separate NUMA node, and manages the NVM nodes independently providing on-demand allocations and free space management. Applications can allocate explictly using library allocators for persistent (nvmalloc), or nonpersistent (malloc) use. Further, pVM offers a set of NVM policies such as 'nvmrevert', 'nvmpreferred' (discussed in Section 4 by extending the OS (memory scaling policy component), and providing support for existing user-space NUMA libraries. pVM also provides explicit control for applications and allocators to directly allocate to NVM with an NVM mmap (nvmmap). We describe the design and implementation details in Section 4.

Next, (2) for fast persistent storage, applications use a persistent library allocator similar to [19, 40]. Figure 3 shows the programming model for persisting an image object to NVM. pVM extends the VM subsystem by providing

object-based persistence that maps each persistent object to a set of NVM pages, with ACID guarantees for OS and application data. Every object is identified using an unique objID which is a combination of a unique global ID per application (gUID) plus the hash of an object name. Hence, applications sharing an object should use the UID as a capability, similar to a shared memory implementation in Linux. Our current implementation provides basic functionality such as object creation, updates, deletes, renaming, write protection, history of object updates, logging and durability guarantees.

Finally, pVM achieves (3) improved performance in part due to its design approach which extends the virtual memory subsystem and its cache efficient data structures and mechanisms.

Limitations. pVMs object store is a good fit for applications that do not rely significantly on the complex filesystem hierarchy, and supports only a flat namespace or one that can be easily modified. Examples include persistent key-value store, photo stores, etc. Hence, **pVM is not a replacement for file-systems**, and does not support all extensive and complex filesystem security policies such as link/unlink, aliases, group permissions. Further, due to its flat namespace, similar to several other prior work [19, 40], the gUID (app UID + object name hash) model prevents multiple object in an application to have the same name, thus avoid name collisions. Further, application sharing objects should implement their own concurrency and transactional mechanism. Our ongoing work is focused on addressing this issue with a shared memory consistency protocol.

4. Design and Implementation

The pVM system comprises of (1) a VM-based OS NVM manager (pVM-OS), and (2) a user-library (pVM-lib) – an interface between pVM-OS and applications.

4.1 pVM-OS support

We first discuss the hardware abstraction of NVM from the view of an OS, followed by the OS support for exploiting NVM for additional capacity and storage use. pVM provides a cleaner abstraction of NVM memory, and more importantly, improves the processor cache and TLB use by extending the VM subsystem, which has undergone decades of optimization, instead of relying on a VFS-based filesystem.

4.1.1 pVM OS Hardware abstraction

NVM as a NUMA node. pVM-OS treats NVM as another NUMA-node, providing support for an OS node-based abstraction for NVM (see Figure 4), with its own NVM-specific memory allocation and management policies. By enabling a node-based abstraction, pVM provides a cleaner hardware abstraction at the OS and application-level, without requiring significant changes across the system stack to leverage existing memory placement mechanisms and tools. This enables pVM to easily extend or add NVM-specific policies and management methods, and supports scaling across multiple NVM nodes in large memory platforms, including benefits like memory hot-plugging. For development and evaluation of such support in the OS, because byte-addressable NVMs are not commercially available, we treat a DRAM socket as an NVM node, using special flags that isolate the NVM from general purpose allocations, and only allocate pages in response to the pVM OS manager. In addition, to differentiate the regions of memory used for capacity, persistent storage, and OS-level persistent metadata and log for object stores, we extend the Linux memory zones with new NVM capacity, persistence, and kernel zones, and discuss shortly how pages are allocated in these zones.

pVM NUMA policies. The benefits of fine grained memory placement flexibility across multi-socket NUMA platforms at application/library/OS has been well studied, along with benefits of data locality. Such memory placement flexibility and policies are even more significant when using NVM for additional capacity due to NVMs' significantly lower bandwidth, high write latency, but larger capacity. Prior solutions [19, 22, 40] that allow use of NVM for capacity, do not provide such memory placement flexibility. By extending the VM-subsystem to a NUMA-based NVM design, pVM can easily add such policies. We added three policies, $nvmrevert$,$nvmpreferred$, and $nvmbind$ to existing user-level NUMA libraries. As the names suggest, $nvmrevert$ reverts to using NVM when all DRAM pages are exhausted, $nvmpreferred$ refers to using NVM as a default node, and $nvmbind$ restricts application to only using NVM. We evaluate the benefits of such policies for memory capacity intensive applications.

4.1.2 pVM OS software abstraction

While a NUMA-based configuration provides a cleaner hardware abstraction, it is important to further specialize for NVM OS data structures across different layers of the VM subsystem. Figure 5 shows multiple layer of virtual memory starting with the OS interface memory-access layer at the user level that provides the mmap/nvmmap interface, followed by the memory mapping layer that decides how a memory region should be mapped to a user address space, and finally the low-level allocator. We first provide background of existing process-related VM datastructures, using Linux for concreteness, followed by changes necessitated at each layer to enable NVM support.

VM background. In OSes such as Linux, each process has an OS context, an associated memory structure (task mm_struct) which encapsulates the entire user address space (heap, code, data and stack segment). A process address space can contain one or more contiguous memory regions known as virtual memory area regions (VMA). All pages in a VMA have same access permission, and are used for similar functionality such as additional heap, code pages, etc. The VMA/regions are created/merged when applications map memory to address space using "mmap" or "sbrk". The

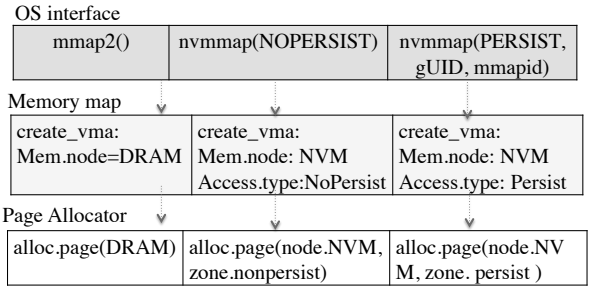


Figure 5: Extension of VM layers to support pVM

pages are added to a VMA only on the first touch (a minor page fault). These VM structures (process context, VMA, and pages) are also important for building an OS persistent state of an application.

Persistent vs. nonpersistent OS interface. pVM exposes applications/libraries with a nvmmap() interface to explicitly map NVM for nonpersistent (capacity) or persistent use into application address space, as shown in Figure 5. The system call signature is similar to the mmap interface, except that additional flags PERSIST vs. NOPERSIST are used to distinguish capacity vs. persistent use. Further, for persistent use, additional application and VMA naming arguments are supplied by the user-level object store library, discuss shortly.

Mapping NVM into address space. The memory mapping layer is responsible for creation of and updates to process memory and VMA structures. For cleaner partitioning of NVM and DRAM datastructures, it is important to differentiate NVM and DRAM nonpersistent datastructures, and similarly, NVM persistent and nonpersistent datastructures. VFS-based NVM mechanisms do not differentiate among persistent and nonpersistent NVM regions, thus imposing metadata bookkeeping penalties for both capacity and persistent storage page access, as shown in our evaluation (Section 5). To enable this, as shown in Figure 5, pVM introduces special VMA memory type flag to differentiate DRAM and NVM region, and an access type flag to further distinguish persistent and nonpersistent NVM regions.

NVM page allocation and per-CPU list. The OS allocator is the heart of any VM-based management. OSes such as Linux use the buddy allocator that has undergone decades of research and optimization. pVM aims to reuse and exploit most of the logic and optimizations, without sacrificing the goal of cleaner separation between DRAM and NVM structures. Hence, we use the VMA memory type and access type to differentiate page allocation requests, and redirect allocations to specific persistent or nonpersistent NVM zones. Further, for persistent allocation, the allocator is also responsible for verifying if the requested page already exist but is not mapped to a process address space, and adds it to the page table. Further, to reduce the overhead of allocator complexity, OS allocators maintain per-CPU free page list as a fast path for allocation and reclamation, before requesting the 'buddy' allocator. However, currently this is limited to homogeneous memory only. We extend the per-CPU list with an array of per-CPU list, containing a separate list for DRAM, NVM persistent, and NVM non-persistent (capacity) allocations. After a page is allocated from a persistent list/zone, the allocator does not have control over the pages, and such pages are reserved, non-swappable, and managed by the OS persistence manager, until released by an application with appropriate permissions.

Configuring NVM persistence vs. capacity use size. The knowledge about space utilization of capacity and persistence use, and options to configure them is necessary for a user to make best use of NVM device for capacity and persistence. Hence, pVM provides users with an option to reserve the capacity of persistent and non-persistent zones (currently via a script), and stats similar to other systems such as PMFS. However, reserving the capacity does not require allocation and addition of pages to page table. In future, we plan to explore dynamic partitioning mechanisms.

4.1.3 pVM-OS persistence support.

Providing support for persistent object stores requires OS and user-level library support. The OS is responsible for persisting all the data structures such as process context, VMA, and pages that form a mapped region, whereas the persistence of the actual content, i.e., application data, is responsibility of a user-level library. In other words, the OS provides metadata persistence for mapped regions, whereas, the user-library manages/uses the region to provide applications with a persistent object store. We first discuss the OS support for persistent mapped region, and discuss the user-level object store support in the following section.

Complexity of persisting VM structures. When using the VFS, the filesystem and its metadata already support persistent mmap regions. In pVM's VM-based design, three important OS structures that form a mapped region require persistence: the process context, all the VMAs inside an application, and finally all page structures inside each VMA. However, persisting these in-memory data structures is significantly complex. This is because, each of these structures dereference other complex OS structures, they are continuously updated by different OS subsystems, and importantly, these data structures are designed for volatile DRAM, and hence are not persistence friendly.

Key Idea: Replica log structures To overcome this, we propose a log-based approach, where we create simplified replica of the required process context, VMA, and page structures. The replica contains only the important information that is required to locate and load all persistent pages of a process. We term the replicas nvproc, nvma, nvpage corresponding to their in-memory process context, VMA, and page structures respectively. Figure 6.(A), shows the in-memory state of a process context with VMAs and related pages. Figure 6.(B) in the middle shows the corresponding NVM replica structures (nvproc, nvma, nvpage) in a tree representation. Figure 6.(C) represents the persistent storage log format of the replica structures.

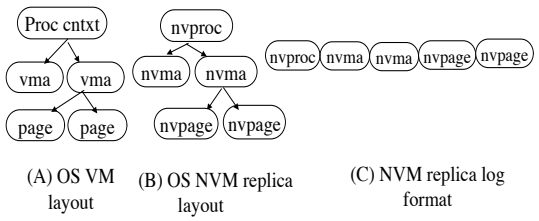


Figure 6: pVM OS persistent structure layout

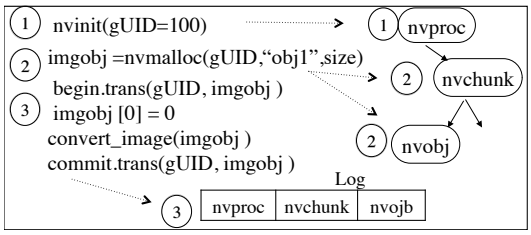


Figure 8: pVM object creation & updates

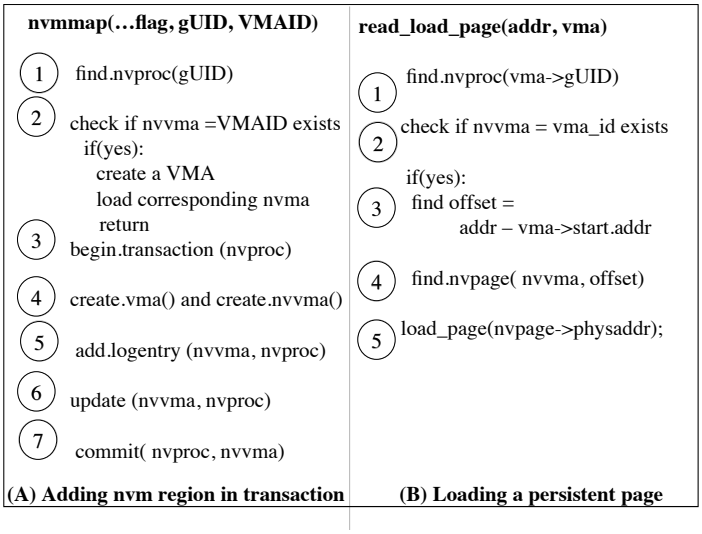


Figure 7: pVM persistent region creation, and page load

Each process context replica, nvproc, can have a tree of nvma, and each nvma can have a tree of nvpage. Each nvproc is identified by a globally unique ID (gUID) supplied by the user library that maps a region of NVM using the OS interface nvmmap. Each nvma contains a start and end physical address corresponding to the virtual address range of the VMA, and a locally unique VMAID supplied by the user-library (generally an incremental mmap counter). The VMAID is used as an index into the process "nvma" tree. Similarly, the nvpage has a corresponding physical address of the page, and an offset from the starting physical address of its VMA. We next discuss the creation and reload/recovery of a persistent mapped region.

Creating and reloading persistent mapped regions . A persistent mapped region is created by an application or an object store library using the OS nvmmap system call. The library also provides a persistent map flag and a globally unique ID (gUID) for each application. pVM's OS-level persistent manager uses the gUID to locate the persistent state of the replica process context in the log, and if not found, creates a new context. Next, a VMA with appropriate persistent flags are created along with their nvma, and added to the replica nvproc's nvma tree, as shown in Figure 6.B. For new persistent pages allocated and added to the in-memory VMA structure, a corresponding nvpage is created and added to the corresponding nvma RB-tree, indexed by the page offset. Figure 7.A,B provides the high-level code for creating a new region, and reading/reloading a persistent region.

To reload/read a persistent regions, the application/user-library provides the gUID and the persistent region VMAID maintained in its user-level persistent metadata (explained below). The gUID and VMAID are used to locate the state of the corresponding region in the persistent replica log and load them into a tree structure. The pages are loaded to application address space only after an application touches/access a persistent page in the mapped persistent region. During the first touch, a page fault is generated, and by finding the offset of the faulting address with the starting address of a VMA, a corresponding nvpage structure is identified which contains the physical address of the page to be loaded/added to applications page table. We favor this lazy approach, as opposed to one which loads all pages at once, in order to reduce restart or read time, and to limit TLB pollution.

Consistency and durability . The OS-layer is only responsible for maintaining consistency and durability of OS persistent state, i.e., the replica structures required to construct the persistent region, whereas the consistency and durability of actual application content is handled by the user-level object store. pVM heavily borrows the ideas and code from PMFS's [22] highly optimized OS-level UNDO journaling with atomic commits for cache-level updates. While PMFS maintains the consistency and durability of the filesystem metadata, pVM maintains it for replica structure. Further, a key difference between the PMFS' filesystem and pVM's VM-based design is that, unlike a single journal for the entire filesystem, pVM maintains a separate journal for each gUID or process replica structures, with a global master journal bookkeeping the location of individual process replica journal. Having multiple journals, specifically for frequent page updates avoids contention for single journal lock across applications. However, pVM currently requires applications/libraries to handle their own object sharing, and concurrent update mechanism. We plan to extend a more transparent mechanism from prior research [42].

Figure 7.A shows the high-level transactional journaling code for creating a new nvma structure. For updating a nvma, the nvma and its corresponding nvproc structure is logged, and for adding/updating a nvpage, the nvpage and nvma are logged. Each log entry in a journal is 64-bytes, and the header and data of a log entry are committed first, followed by a the log tail. Updates to the log are ordered

with optimized write barriers (WB_BARRIER instruction) and cache flush (CL_FLUSH), discussed in detail by prior work [22]. Recovery happens by first reloading the master journal, followed by the log data in the journal. pVM follows an all-or-nothing model to guarantee ACID property for OS persistent state. Hence a failure to load any one persistent structure marks the entire application state unusable.

4.2 pVM-lib allocator and object store

We next discuss pVMs user-level library that provides application with transparent NVM allocator for capacity use, and object store support.

pVM capacity allocator and NUMA policies. Modern allocators map large regions of memory from the OS, and manage them for subsequent allocations. For pVM, we extend the scalable jemalloc [8] library allocator with nvmmap support for directing allocations to NVM. The jemalloc library allocator is used for both nonpersistent capacity and persistent allocation with appropriate flags to classify them. By using a library allocator, no application-level changes are required for nonpersistent applications. Further, applications can also use the NVM NUMA policies.

pVM object store interface pVM's object store interface and ACID mechanisms are inspired from prior NVM-as-heap research [19, 40] to create named objects. We first briefly describe the application interface followed by its object store metadata management customized for VM-based OS design. The object store provides applications with a 'nvmalloc' interface to create named, uniquely identified object object (objID) formed by combining the per-application UID and the unique object name.. Figure 8 shows a sample code to allocate an image object, and use the reference of the allocated object to update the object via a byte addressable load-store interface. To provide consistency and durability, pVM uses the Intel's persistent memory library [29], to wrap updates to an object inside transactions. We discuss the details next.

Persistent object store metadata. An object store creating persistent objects should also maintain persistent metadata about the objects such as their location in mapped regions, size, objID, and information about consistency. Because modern allocators are complex, and maintaining the entire state of allocators in NVM can be expensive, we create a simplified replica structure in a log, similar to that for OS-level persistence. During application initialization, a unique gUID is generated, and a corresponding user-space "nvproc" structure is created in a user-level persistent log as shown in ① in Figure 8. Next, the allocator creates a large persistent region with gUID, and an incrementing MMAPID (used as VMAID by the OS). Upon successful creation of an NVM region, a corresponding "nvchunk" structure is added to the NVM metadata as in ②. Because each application can have several mmap'ed regions, each nvproc can have several nvchunks. Next, for every object mapped to a persistent region, a corresponding nvobject is added to the nvchunk.

Capacity Apps	Description	Workload	NVM usage type
FaceRec [4]	OpenCV-based face recognition using gallhager [23] dataset	1.2GB input DB (2K images, each 800KB)	Capacity
Metis [16]	Uses Metis with 4 mappers-reducers	2GB crime dataset	Capacity
GraphChi[30]	Graph pagerank algorithm	Orkut social graph, 3 million nodes, 117 million edges	Capacity
Dedup [15]	Parsec Dedup benchmark	4GB OS image file	Capacity
Snappy [6]	Fast data compression used in chrome and other Google products	Image, video, audio, document files -2GB	Object store
LevelDB [24]	Google's DB used from browser to datacenter	SQLite benchmark 500K operations	Object store
Phoenix [44]	Shared-memory mapreduce running word-count	same as Metis	Object store

Table 2: Applications

To provide consistency and durability, the objects and object store metadata are updated within transactions, using an UNDO log ③. Note that the UNDO log is a journal located separately from the persistent metadata. Our consistency and durability mechanism is borrowed from object-based logs of prior work [19]. For reading/recovering a persistent object, application use the nvread() interface, with the gUID and object name. The object ID generated from the combination of gUID and object name is used to locate the corresponding persistent region, and instruct the pVM's OS manager to load the region to application address space.

Discussion. Several research proposals have focused on optimizing NVM object stores over VFS. However, pVM's main contribution is its generic VM-based OS (pVM-OS) design that addresses capacity bottlenecks, and improves object storage performance. To understand the effort required to adapt other open source object stores with pVM-OS design, we extended the SNIA-based NVML library [29] from Intel, with 75 lines of code changes. NVML maintains its own object store metadata, hence, to create pVM-based persistent regions, we replaced the the VFS-based mmap to nvmmap, and map the objects to the corresponding nvmmap regions using the unique ID for the mapped region similar to pVM's gUID.

5. Experimental Evaluation

We evaluate pVMs (1) capacity scaling capability, and its impact in improving performance, (2) cache, TLB usage efficiency, and page-allocation/access time, (3) object storage performance and its implications on the time spent inside OS I/O stack, and finally, 4) the cost of consistency and durability guarantees. To evaluate, we study several benchmarks, and applications listed in Table 2. We next discuss the experimental setup, the NVM emulation method, followed by the evaluation.

Platform	Intel-Nehalem platform, 2.4 GHz, 8 CPUs
Cache	12MB LLC
DRAM	6 GB DDR3, Dual socket
Storage	Intel-510 Series SSD
OS	Linux 3.9.0 with PMFS support
pVM setup	3GB DRAM - 3GB pVM's NUMA-based node
pmfs setup	3GB DRAM - 3GB VFS-based PMFS on second socket

Table 3: Experimental setup

5.1 Methodology

Experimental Setup and NVM Emulation Table 3 shows the system configuration for our evaluation. We use a Intel-Nehalem based platform, with one of the NUMA socket emulated as pVM. Next, for emulating NVMs to 10x lower B/W, 2x, 5x slower read and write, we first reduce the the bandwidth of DRAM socket up to 10x relative to DRAM using thermal throttling of a NUMA socket. Next, for emulating latency, we dynamically inject delays in the application runtime by finding the number of load/store LLC misses after every epoch. While Dulloor et al. [22] use a similar technique on the processor microcode-level in a server-class machine, due to lack of access to such microcode, we emulate this in S/W similar to prior work [40]. For our experiments (capacity, and object store), we consider three configurations shown in Table 3, ① DRAM-only - a 6GB DRAM only configuration, ② PMFS - split 3GB DRAM, 3GB managed by PMFS, and finally, ③ pVM - split 3GB DRAM, 3GB NUMA node managed by pVM.

5.2 Applications use of NVM

We use a set of benchmarks (discussed with results) and the applications shown in Table 2 for evaluating the application memory capacity scaling and object store benefits and implications of pVM. Table 2 classifies the applications based on their NVM usage type. For applications classified as 'capacity', we use NVM only for capacity scaling (I/O happens to SSD), whereas 'object store' applications use NVM for persistence and additional capacity. For using PMFS for additional capacity, we use Intels NVML library [29] designed to work with PMFS, supports non-persistent capacity allocations using scalable jemalloc allocator [8] that maps memory from PMFS using MAP_PRIVATE flag [3]. For explicit allocations, pVM also uses jemalloc, and additionally, for memory scaling analysis, we use pVM's Linux NUMA library extensions that supports NVM-based NUMA policies, and we use an allocator over PMFS for comparison. For object-store applications, we replaced the POSIX interface with NVMs heap-based object store. Most of these applications (LevelDB, Snappy, Phoenix) are a good fit for object storage as discussed by Harter et al. [28], as they are either not dependent on the file-system hierarchy, or load the files into an in-memory object before processing them. Modifying Snappy [6] required less than 1 man-day.

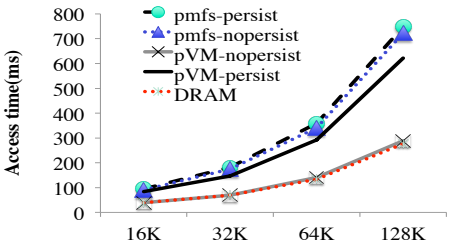


Figure 9: Page access time {persist, nopersist}- indicate persistent (storage) and nonpersistent(capacity) page access

5.3 NVM capacity use analysis

To understand the effectiveness of pVM's VM-based design when using NVM for additional capacity, we first use a set of benchmarks to evaluate page access cost, cache and TLB effectiveness, and then use real-world capacity intensive applications to understand the implications of pVMs NUMA-based policies, and compare them with VFS-based PMFS.

5.3.1 Page allocation and access cost.

Page allocation cost is an important metric for understanding the OS, and application performance. Applications/allocators map regions of memory or NVM with 'mmap' (pVM uses nvmmap) interface for capacity and persistence needs, however the actual allocation of a page happens only on its first-touch. Hence to measure the page allocation cost, we use the Linux scalability benchmark *mmap bench* [2], that first maps a large region of memory and randomly touches mapped pages across their page boundaries. In Figure 9, the x-axis shows the total pages accessed, and the y-axis represents the access cost that includes the cost of allocation also. We analyze this for both persistent (pVM-persist, pmfs-persist), and nonpersistent allocations (pVM-nopersist, pmfs-nopersist).

Analysis. pVM classifies nonpersistent (capacity) from persistent allocation using nvmmap() flags, and hence does not add filesystem metadata/bookkeeping or journal overheads for nonpersistent allocation. However, VFS-based PMFS lacks such classifying capability adding overheads for pmfs-nopersist also. As a result, pVM-nopersist reduces page cost reduces by 2.5x% relative to pmfs-nopersist. Next, for persistent page allocations, for each allocation, a page mapped to a file is located from the filesystem B-tree, loading the dentry caches if required, and updating several complex data structures such as inode, inode bitmap, blocks, and superblock. While pVM has to also locate the pages from VMA-level (nvma) RB-tree, only three simple replica structures nvproc, nvma, nvpage (mostly nvma,nvpage) are updated and journaled. Hence, pVM-persist shows a maximum gain of 19.67% compared to pmfs-persist. The small improvement of pmfs-nopersist over pmfs-persist is from the use of MAP_PRIVATE flags, that instructs the filesystem to skip persisting the page content. Incorporating B-tree instead of RB-tree in pVM can improve the gains further. *The results show the importance of using the using the VM-*

App	Type	WSS
Canneal	Engineering	2GB
Facesim	Animation	256MB
Ferret	Similarity Search	128MB
Fluidanimate	Animation	128MB
Streamcluster	Data Mining	256MB
Swaptions	Financial Analysis	512KB
x264	Media processing	16MB

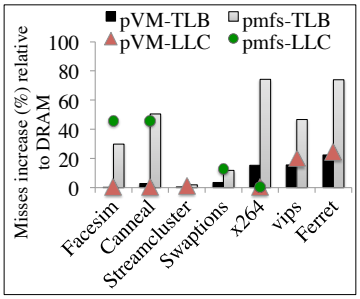


Figure 10: TLB, Cache miss analysis. Y-axis denotes factor(x) increase relative to DRAM

based design and classifying persistent and nonpersistent page allocation/access.

5.3.2 TLB, cache usage efficiency.

We next study pVM's effectiveness towards processor cache and TLB using the well known PARSEC benchmark [15]. We use a wide range of applications in terms of their working set size (WSS), and capacity usage [14] with 'native' (real-world execution) as shown in the table in Figure 10. We use the setup in Table 3. The graph on the right side shows the increase in the TLB and cache (LLC) miss increase factor (in y-axis) relative to using only the DRAM (baseline).

Analysis. As evident from the figure, pVM by extending the VM significantly reduces TLB, and LLC miss up to 68% and 26% respectively, relative to the VFS-based approach of PMFS. Most applications with pVM show less than 3-5% increase in the TLB and LLC misses relative to baseline, except vips, ferrert, and and x264, which showed 10%-22% (ferret) increase. For these applications we observed higher remote node (NVM node) misses. Next, for PMFS, we notice higher TLB misses (up to 70%) specifically when the NVM capacity usage is high (Facesim, canneal) irrespective of the WSS (x264), whereas for applications with large WSS such as Facesim, canneal, the LLC misses are also high. A closer analysis shows that high LLC and TLB misses with PMFS are mainly due (1) the VFS does not classify persistent and nonpersistent pages, hence performs transactional bookkeeping of filesystem metadata for both resulting in VFS cache and TLB pollution coupled with datastructures not designed for cache efficiency. *The results highlight the importance of adopting a VM-based design.*

5.3.3 Memory scaling and placement impact

To evaluate the effectiveness of pVMs memory placement policies in addressing the memory scaling, and performance issues, we use the applications in Table 2 with NVM usage set to capacity. We compare the (1) DRAM only, (2) PMFS, and finally (3) the proposed pVM using the same setup described in Table 3. With pVM, we evaluate two policies 'pVM-nvmrevert (revert to NVM if DRAM is exhausted), 'pVM-nvmpreferred (allocate first to NVM). The input data for the applications are placed in SSD to clearly understand

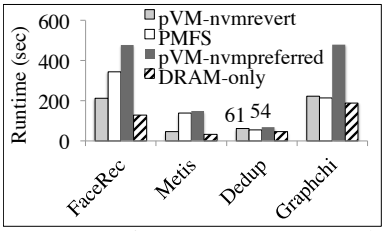


Figure 11: Memory scalability impact.
(pVM-nvmrevert, pVM-nvmpreferred are NUMA policies)

the capacity benefits. Note that, memory-based filesystem including PMFS cannot be used as a swap in Linux.

Analysis. As shown in Figure 11, clearly the optimal case DRAM, and pVM-nvmrevert clearly outperform PMFS. When the DRAM is exhausted, the OS lacks the support to transparently allocate from PMFS backed NVM because of a different managing (VM, and VFS) subsystems. With the increase in DRAM pressure, swapping is initiated resulting in application slowdown. However, pVM-nvmrevert provides a seamless memory scaling capability to automatically switch to NVM when DRAM is exhausted. While using NVM reduces performance relative to DRAM, it still provides 2.5x, and 2x speedup relative to PMFS for FaceRec, and Metis. Interestingly, the Dedup with 3GB peak memory usage and 3GB NVM (a corner case), the PMFS case (54 sec) marginally improves performance by relative to pVM-nvmrevert (61 sec). This is because, the pVM-nvmrevert starts allocating pages from slow NVM once a non-critical DRAM threshold is reached [26] to avoid exhausting all the DRAM resources, a logic borrowed from current Linux NUMA policies, whereas PMFS exhausts all 3GB without using slow NVM pages. A more stricter NVM specific policy can avoid such overhead. Next, as expected pVM-nvmpreferred has the highest overhead. We expect the policy to be useful for low priority or memory bandwidth and latency insensitive applications. As observed for benchmarks and application discussed in Section 2, pVM with its VM-based design reduces cache, and TLB misses by 80%, and 74% respectively, relative to VFS-based PMFS (not shown for brevity), a significant improvement compared to the parsec benchmark gains (up to 26%). *The results show importance of pVM's memory scaling capability via. flexible NVM memory placement policies.*

To analyze the impact of pVM's VM-based object storage performance, we first analyze the performance and the OS-level overheads such as journaling for well-known NoSQL LevelDB [24, 25], followed by the performance, and cache implications of two applications, Snappy, and Phoenix described in Table 2. We analyze four cases, (1) pVM-obj (proposed), (2) pmfs-obj that uses pVM's user-lib for object store, but maps memory from VFS-based PMFS, (3) pmfs-mmap - mmap version of applications instead of block-I/O, and finally, (4) pmfs-block (baseline) - block based file I/O with traditional POSIX interface. To clearly understand the

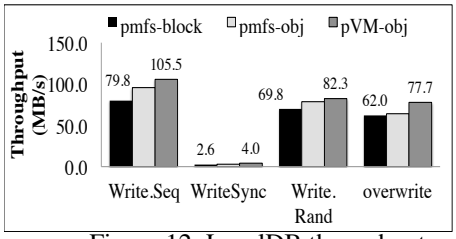


Figure 12: LevelDB throughput comparison

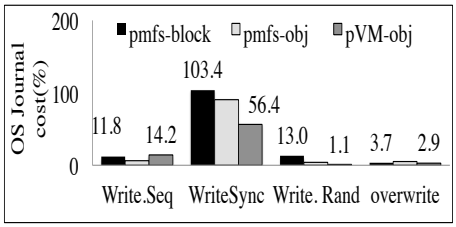


Figure 13: LevelDB: OS journaling cost(%) on throughput

storage performance, we use NVM only for storage, and use DRAM for capacity.

5.3.4 Database benchmark

LevelDB provides applications with a key-value interface and is a perfect fit for object-based store. LevelDB internally uses POSIX block interface for updates, and mmap interface for reads. Figure 12 compares the throughput (numbers over bar are in MB/sec) for pVM, and pmfs-obj relative to pmfs-block for 500 K transactions of the LevelDB [24] benchmark. The x-axis shows the access pattern. For Write-Sync, updates even inside a transaction is committed (logged), unlike Write-Seq, and Write-Random where commits are done only outside the transaction. Because for reads the entire database is mapped to memory without significant performance difference between different approaches, hence we only study the write patterns. Figure 13 shows the reduction in throughput (%) as result of OS-journaling.

Analysis. Clearly, both pmfs-obj and pVM provide higher throughput relative to pmfs-block. pmfs-obj improves throughput for sequential, and synchronous writes by $\sim$15%, $\sim$19% respectively, whereas pVM-obj achieves $\sim$32%, $\sim$53% improvement. Both, pmfs-obj, and pVM-obj gain by reducing the OS-filesystem mediation and lower context switch cost due to frequent I/O calls (write, fsync, flush, stat), reduced user-space to kernel data copy overheads (discussed in Section 2), and importantly lower OS journal cost shown in Figure 13. In the case of pVM vs. pmfs-obj, even though pmfs-obj reduces OS filesystem use and kernel mediation, but because the objects are mapped to a region managed by VFS (i.e., PMFS), we observe higher OS-journal overhead due to complex filesystem metadata, higher page access cost of frequent allocation and release for small transactions, and finally, higher cache, and TLB misses. Because pVM is specialized for object store, the OS journaling is simpler compared to the filesystem, as it needs to maintain a consistent state of only its replica structure with some bookkeeping discussed in Section 4. This reduces the OS journaling impact on the database throughput (see Figure 13).

5.3.5 Object store analysis for applications

We next study the impact of pVM's VM-based design in improving the overall application performance, and the OS time reduction.

Impact of object-based interface: In Figure 14, the left y-axis shows the file compression throughput for Snappy (MB/sec), and the right y-axis shows the time spent inside OS and filesystem. Figure 15 shows the runtime comparison (left y-axis) for a word count on a 2GB crime dataset, and the right y-axis shows the corresponding OS time. First for Snappy, the both pVM-obj, and pmfs-object provide significant gains over the the traditional pmfs-mmap, and pmfs-block. As discussed earlier in Section 2, applications like Snappy show high variability in I/O sizes, and perform frequent I/O calls. Using an object-based interface (pVM-obj, or pmfs-obj) significantly reduces the filesystem kernel mediation time, I/O library cost, metadata and related synchronization overheads of pmfs-block and pmfs-mmap. As a result, the time spent inside OS (and specifically the filesystem) reduces by up to 4x. Interestingly, using the mmap interface with pmfs-block deteriorates performance compared to pmfs-block mainly because of the increased kernel mediation from several supporting system calls (such as fopen, fstat, mmap, and munmap) for files which have low processing time. Regarding, the Phoenix mapreduce [44], it maps the entire input into application address space, and computer over the mapped input either backed by VFS or pVM. So no I/O calls are made after initial mapping, hence object-based interface provides less than $\sim$ 11% improvement over pmfs-block. The mmap mode, performs better compared to the block-mode by avoiding an extra copy from filesystem to DRAM buffer, and has almost similar performance as pmfs-mmap. For the OS time reduction in pVM-obj, we will next discuss the key difference between pVM-obj, and pmfs-obj.

Extending VM for persistence: Next, comparing pVM, and pmfs-obj, pVM reduces the kernel time for Snappy by 2x relative to pmfs-obj, 2x improvement in throughput, and around 14% for Phoenix. The reasons for improvements are as follows. (1) Although pmfs-obj uses the pVMs object library at user level, the objects reside in files mapped and managed by the PMFS file-system including the persistent page allocations and access. As, discussed earlier, pVM by extending highly optimized VM-based design benefits from its faster page access, avoids update complex file-system-based metadata updates, and reduced journaling overhead cost as discussed for LevelDB benchmark. This results in fewer instructions and NVM access. As shown in Table 16, pVM reduces the number of instructions, TLB and cache misses relative to pmfs-obj. Even though seemingly small, due to significant the gains are of significant importance due to 5x higher write latency of NVM relative to DRAM.

Summary: The following conclusions may be drawn from the experimental result shown above. (1) pVM's VM-based design with NUMA capability, addresses the memory scal-

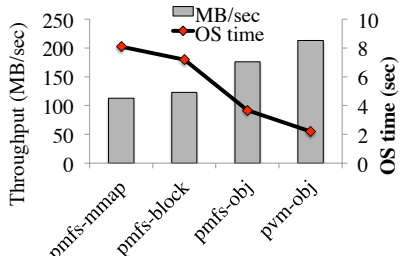


Figure 14: Snappy throughput, OS time

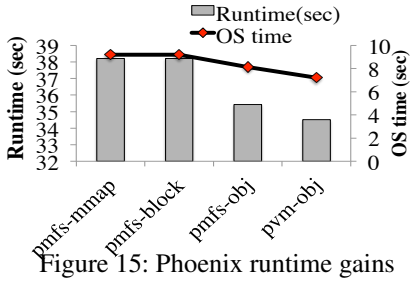


Figure 15: Phoenix runtime gains

App	Instruct (%)	TLB Miss(%)	Cache Miss(%)
snappy	8.20	11.7	6.32
leveldb	9.40	8.32	6.16
Phoenix	3.17	1.1	5.6

Figure 16: pVM cache, TLB miss, CPU instruction reduction relative to pmfs-obj

ing issue providing applications with flexible memory placement policies, and improves application performance by up to 2x. Next, by classifying memory pages for capacity vs. persistence use, pVM reduces TLB, and cache misses by up to 80%, 74% (and 26% for benchmarks). Further, pVM's object store reduces software-stack overheads, page access time, CPU instructions, cache, and TLB misses resulting in $\sim$2x higher throughput, and up to 4x reduction in OS time relative to block I/O, motivating the the need for OS redesign for object storage.

6. Related Work

We first discuss the NVM capacity scaling research, followed by the persistence storage related research.

Capacity scaling in hybrid memory systems. Prior research, such as [21, 31, 35, 38] use NVM as an alternative to DRAM with transparent page replacement strategies between DRAM and NVM. Qureshi et al. proposed a hardware-based model that treats DRAM as a cache, and NVM as main memory, and use the page access patterns in the OS/hardware to move data between DRAM, and NVM. While this provides application/OS transparency, copying data across between a fast and slow memory can add significant overhead. It limits application flexibility in memory placement, and importantly prevents the use of persistence property. Saxena et al. [38] propose using flash as an extended memory managed by VFS, and use several data prefetching techniques. PMFS also enables extending memory capacity via. VFS. In contrast, pVM provides a VM-based management for enabling seamless memory scaling, and also provides flexible memory placement mechanisms by treating NVM as a NUMA node. To the best of our knowledge pVM is the first OS-design that considers extending VM for both capacity and persistent storage, and provides applications with flexible memory placement policies.

Filesystem for NVM. Next, regarding persistent storage, Aerie [41], Moneta-D [18], and some early work such as FUSE [36], aim to reduce the OS overhead, by moving filesystem to the user space, and directly using hardware for read/write operations, with flexible I/O interface. We discussed the decentralized design of Aerie in Section 2. Other research such as [17, 20, 22, 43] have proposed optimization to existing block-based filesystem, and adapting them for memory-based persistence that were discussed earlier. pVM borrows several OS consistency and durability-related ideas, and adapts them for object storage. Our future work will consider the co-existence of hierarchical filesystem with pVMs object based storage.

NVM as a heap. The benefits of using NVM as a heap for persistent storage has been well documented by one of the earliest systems Rio Vista [32], and RVM [37]. they provide heap-based persistence, and the OS-level management that uses a combination of block-based storage and battery-backed RAM. Further, Rio Vista provides persistence guarantees only at log flushes. Recent research such as [19, 40] also offer persistent data programming models in addition to capabilities of RVM. pVM's user-level library borrows ideas from these proposals. However, the key contribution of pVM is the novel extension of the VM subsystem for memory scaling and persistence. [19, 40] depend on VFS-backed memory, and limitations have been discussed extensively. Further, Volvos et al. [40], and [43] propose a NVM persistence OS zone. pVM extends such ideas to create capacity and persistence zone, but importantly, provides a cleaner abstraction by treating NVM as NUMA nodes, and providing users with flexible memory placement policies. Guerra [27] et al. discuss solution for addressing mixed persistent and nonpersistent pointers via. memory protection, which can be adapted to pVM also.

7. Conclusions & Future Work

We propose pVM, a OS mechanism that extends byte addressable NVMs for exploiting capacity and storage benefits. pVM integrates NVM into the OS as memory (NUMA) nodes and extends the virtual memory to manage NVM nodes instead of the VFS. By doing so, pVM addresses the memory scaling issues of state of the art design and provides 'close to hardware overheads',. Further, pVM provides users with flexible memory placement support which is integrated with existing user-level NUMA libraries. pVM further extends the VM subsystem for fast object-based storage to overcome substantial file-system bottlenecks. By further distinguishing capacity and persistence access at the OS, apart from the performance benefits discussed in this paper, we believe there are several opportunities to bridge the gap between volatile memory use and persistent storage. While we evaluate pVM with several data-intensive applications, we plan to explore more end-user centric environments, such as Android stack for gaming, multimedia and other applications.

Artifacts. In order to preserve anonymity, we cannot distribute source code at this time. If the paper is accepted, we will open source the technology.

References

[1] Intel-Micron Memory 3D XPoint. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html.

[2] Linux Scalability Benchmark. http://lse.sourceforge.net/.

[3] NVML nonpersistent allocator. http://bit.ly/1RZf4aM.

[4] OpenCV. http://opencv.org/.

[5] S3fuse filesystem. https://github.com/s3fs-fuse/s3fs-fuse.

[6] Snappy Compession. http://tinyurl.com/ku899co.

[7] CEPH file-systems. https://ceph.com/ceph-storage/file-system/.

[8] http://www.canonware.com/jemalloc.

[9] Numonyx PCM projection. http://bit.ly/1sph5Qz.

[10] Samsung 8GB chips. http://bit.ly/1oFFiSR.

[11] SNIA persistent memory specs. http://tiny.cc/o55p4x.

[12] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: A protoype phase change memory storage array. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2002218.2002220.

[13] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 707–722, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. . URL http://doi.acm.org/10.1145/2723372.2749441.

[14] A. Bhattacharjee and M. Martonosi. Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 29–40, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3771-9. . URL http://dx.doi.org/10.1109/PACT.2009.26.

[15] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton, NJ, USA, 2011. AAI3445564.

[16] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855745.

[17] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4299-7. . URL http://dx.doi.org/10.1109/MICRO.2010.33.

[18] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. . URL http://doi.acm.org/10.1145/2150976.2151017.

[19] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1. . URL http://doi.acm.org/10.1145/1950365.1950380.

[20] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. . URL http://doi.acm.org/10.1145/1629575.1629589.

[21] G. Dhiman, R. Ayoub, and T. Rosing. Pdram: a hybrid pram and dram main memory system. DAC '09, New York, NY, USA.

[22] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. . URL http://doi.acm.org/10.1145/2592798.2592814.

[23] A. C. Gallagher and T. Chen. Clothing cosegmentation for recognizing people. In *CVPR 2008*.

[24] Google. LevelDB. http://bit.ly/1wEklYD, .

[25] Google. LevelDB benchmark. http://bit.ly/1ERE4u7, .

[26] M. Gorman. Understanding the Linux Virtual Memory Manager. http://bit.ly/1n1xIhg.

[27] J. Guerra, L. Marmol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei. Software persistent memory. In *USENIX Annual Technical Conference*, pages 319–331, 2012.

[28] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the i/o behavior of apple desktop applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 71–83, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. . URL http://doi.acm.org/10.1145/2043556.2043564.

[29] Intel. Logging library. https://github.com/pmem/nvml.

[30] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6.

URL http://dl.acm.org/citation.cfm?id=2387880.2387884.

[31] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. . URL http://doi.acm.org/10.1145/1555754.1555758.

[32] D. E. Lowell and P. M. Chen. Free transactions with rio vista. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 92–101, New York, NY, USA, 1997. ACM. ISBN 0-89791-916-5. . URL http://doi.acm.org/10.1145/268998.266665.

[33] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-ordering consistency for persistent memory. In *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pages 216–223, Oct 2014. .

[34] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for nvm+dram hybrid main memory. HotOS'09, Berkeley, CA, USA, 2009.

[35] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA '09*.

[36] A. Rajgarhia and A. Gehani. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 206–213, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-639-7. . URL http://doi.acm.org/10.1145/1774088.1774130.

[37] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Trans. Comput. Syst.*, 12(1):33–57, Feb. 1994. ISSN 0734-2071. . URL http://doi.acm.org/10.1145/174613.174615.

[38] M. Saxena and M. M. Swift. Flashvm: Virtual memory management on flash. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 14–14, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855840.1855854.

[39] D. Sengupta, Q. Wang, H. Volos, L. Cherkasova, J. Li, G. Magalhaes, and K. Schwan. A framework for emulating non-volatile memory systems with different performance characteristics. In *Proceedings of the Ninth European Conference on Computer Systems*, ICPE '15. ACM, 2015.

[40] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1. . URL http://doi.acm.org/10.1145/1950365.1950379.

[41] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. . URL http://doi.acm.org/10.1145/2592798.2592810.

[42] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *Proc. VLDB Endow.*, 7(10):865–876, June 2014. ISSN 2150-8097. . URL http://dx.doi.org/10.14778/2732951.2732960.

[43] X. Wu and A. L. N. Reddy. Scmfs: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. . URL http://doi.acm.org/10.1145/2063384.2063436.

[44] R. M. Yoo et al. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. IISWC '09.

HeteroMem: Coordinated Management of Heterogeneous Memory in Virtualized Systems

Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan, CERCS, GATECH

Abstract

To address the 'memory wall' problem of future datacenter servers, vendors are creating heterogeneous memory structures, supplementing DRAM with on-package stacked 3D-RAM and high capacity non-volatile memory. This, along with server virtualization, presents challenges to operating systems' memory management. This paper explores the management of heterogeneous memory in virtualized environments. Its novel approach, termed Heteromem, (i) exposes heterogeneity to guest OSs and (ii) provides an infrastructure for coordinated VMM-guest level hybrid memory management. The approach is motivated by the limitations of purely VMM-level approaches that fail to leverage the existing guest OS-level information, leading to high overhead reactive management strategies. Heteromem, on the other hand, exploits the rich OS-level virtual memory information such as different types of pages, their current state, and other information to proactively allocate memory 'right', thereby reducing the need for memory migration, and, when proactive allocation is not possible, it uses OS-level hints for guiding VMM-level resource monitoring and cross-memory node data movement more efficient. By using a node-based OS design, HeteroMem also permits integration of existing memory ballooning mechanism for fair-share-based heterogeneity management across collocated guest VMs. Its evaluation with well-known large scale cloud applications shows up to 2x performance improvements compared to VMM-exclusive management.

1. Introduction

The combined rapid increase in transistor density, platform-level parallelism, and use of large datasets in modern scientific and commercial workloads is exerting severe capacity and bandwidth pressure on server memory systems. This has led to the exploration and use of alternative memories supplementing traditional DRAM, including byte-addressable nonvolatile memories (NVMs), and on-chip stacked 3D-DRAMs. NVMs such as PCM can provide 2-4x higher capacity than DRAM, but 5x lower bandwidth, whereas on-chip 3D-DRAMs are expected to increase bandwidth by 8-14x [13], but do not scale in terms of capacity [13, 32, 10, 22] and energy constraints [12, 32].

Recent research has considered the unique needs of individual memory technologies. 3D-stacks have been proposed as last-level cache [48, 24, 34, 31, 12], or as fast OS-managed 'NearMem' [22, 13, 16, 10], whereas, NVM has been considered for fast I/O devices or for slow memory [43, 14, 35, 27]. Yet there has been little work to understand how future operating systems should manage these complex memory structures: where to place which data, how or when to move it, and how to do so in the typically virtualized settings of cloud and datacenter workloads. Specifically, in virtualized environments, the VMM has direct access to and is responsible for allocating, managing and controlling the shared use of these heterogeneous memory resources, on behalf of and often across multiple, guest OSs. VMM-exclusive solutions [22], as also done when managing NUMA domains in virtualized servers [38] have limited information about how guests and their applications use memory, and hence, must depend on reactive placement strategies that track use of *all* memory pages of guest virtual machine (VMs), then move hot pages to nearer memory [22, 32]. They, therefore, fail to understand the sensitivity of application performance to the characteristics of the underlying memory, which we demonstrate for several classes of popular cloud and datacenter workloads. Guest OSs, however, already maintain rich information that can better categorize how memory is used, and therefore, better guide how such memory should be managed. Further, future OS are likely to have native support for heterogeneous memory [39, 17] as current NUMA support in VMs [9] is not fully sufficient [32].

The insights above lead us to propose HeteroMem – a solution for managing memory in virtualized systems that (1) exposes memory heterogeneity at the guest OS-level, and (2) provides an infrastructure for coordinated VMM-guest OS management. Using Heteromem results in significant improvements in application performance and in the overall efficiency in using underlying heterogeneous memory resources. HeteroMem achieves guest-level heterogeneity-awareness by extending the existing virtual memory design to support on-demand allocation and memory management. This enables it to exploit the rich OS-level information about how applications use memory, or which pages can have critical impact on application performance. HeteroMem first uses this information to pro-actively allocate memory 'right' in the first place, thereby reducing the need for future cross-memory migrations. Second, when such proactive allocations are not possible, e.g., due to limited NearMem capacity, HeteroMem uses existing guest OS-level information to identify and promote critical/active pages closer to NearMem, thereby improving overall application performance. When such OS-level information is insufficient, the guest-OS guides the VMMs to actively monitor memory regions to identify where management operations can be most effective. Such guidance reduces monitoring overheads and improves the efficiency of data migration.

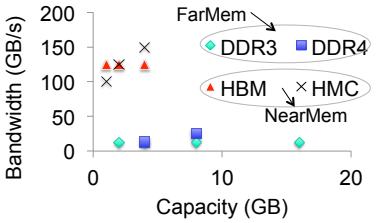


Figure 1: Heterogeneous Memory Properties

HeteroMem design principles are not focused purely on in-memory applications, but is applicable to arbitrary cloud and datacenter workloads, including I/O- (storage and network) intensive applications. This paper concerns its use two levels of memory, (1) capacity limited, high bandwidth on-chip 3D-DRAM (NearMem), and (2) off-chip higher capacity (see Figure 1), lower bandwidth DDR-based DRAM (FarMem). HeteroMem's analysis and design principles can be extended to also consider other classes of memories. The technical contributions of this paper are as follows:

- **Memory heterogeneity analysis, VMM-only management, VMM-guest OS coordination** (Section 2): a detailed analysis of the impact of memory heterogeneity, in terms of bandwidth, and capacity, for cloud-based applications running in virtualized environments, demonstrating the limitations of VMM-only management, and motivating coordinated VMM-guest OS methods.
- **Guest OS memory heterogeneity support:** (Section 4) Guest-OS heterogeneous memory (aware) management that extends their existing virtual memory designs, permitting each OS to directly manage its memory resources. This means that HeteroMem's design applies to traditional non-virtualized OSs, as well.
- **HeteroMem** (Section 4, 5): an architecture and approach to using heterogeneous memory in virtualized platforms that leverage *coordinated VMM-guest OS management*, combining guest OS information with VMM hardware control.
- **Management principles engendering efficiency** (Section 3): HeteroMem derives efficiency from using basic principles that reduce data movement (i.e., migration) and the movement overhead across heterogeneous memory.
- **Performance evaluation** (Section 6): analysis and evaluation with cloud-based applications with varying CPU, memory, and I/O intensiveness, and a study of implications for various heterogeneous memory management techniques. Experimental results demonstrate up to 2x performance improvement compared to VMM-exclusive management.

2. Motivation and Background

Ongoing trends with data-centric applications pose increased demand on the memory systems. At the same time, scaling issues with DRAM are forcing architects to redesign systems with heterogeneous memory devices. While devices such as phase change memory (PCM) and STTRAM are expected to address the capacity bottlenecks, they suffer from lower bandwidth, higher latency, and lower lifetimes. In contrast,

Application	Description and Workload	Performance metric
GraphChi [29]	Graph processing, uses Orkut social graph input, 3 million nodes, 117 million edges, requires 8 GB memory [19]	time(seconds)
XStream [40]	Edge-centric graph processing and uses same input as GraphChi	time(seconds)
Metis [11]	Shared memory mapreduce that optimizes Phoenix [36], 2GB crime dataset, 8 mapper-reducer threads	time(seconds)
LevelDB [1]	Google's DB for bigtable, SQlite bench with 1M keys	throughput(MB/s)
Redis [7]	Popular in-memory key-value store with support for persistence, Redis benchmark with 2M operations, 80% get	requests/sec
NGinx [5]	Popular webserver, 1 million static,dynamic, images webpages	requests/sec

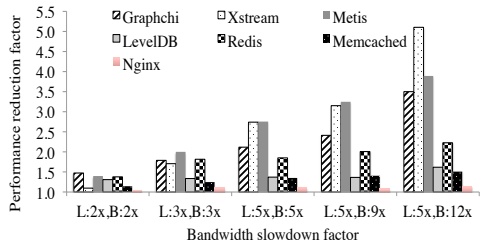
Table 1: Applications

technologies such as Hybrid Memory Cube (HMC), High Bandwidth Memory (HBM) [13, 32], and Wide I/O(WIO), provide high bandwidth and DRAM-like memory array access latency – and are thus referred to as NearMem technologies (see Figure 1) – but are projected to have limited capacity (1GB-8GB) compared to commodity DRAM (i.e., DDR3, and DDR4) [13]. This makes it difficult to completely replace commodity DRAM. In summary, *future platforms will seek benefits by combining different types of memory technologies, and thus integrating heterogeneity in the memory subsystem requires accompanying changes in the software stacks responsible for their management.*

2.1. Impact of memory bandwidth and capacity

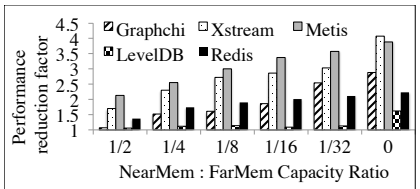
To provide effective management of heterogeneous memory, we first seek to understand the implication of memory heterogeneity, by analyzing the memory bandwidth and followed by the impact of capacity for cloud-based application. Unlike prior research [32, 12, 22] which studies only CPU and memory intensive benchmarks, we include CPU, memory, storage and network intensive applications such as graph analytics, in-memory data stores, map-reduce-based computations, as well as traditional databases and webservers (see Table 1) [19]. The analysis is based on the dual-socket 16-core, 16GB virtualized platform, described in greater detail in Section 6. To emulate near stacked 3D-DRAM with 10x higher B/W than DRAM, we use MSR register-based thermal throttling [41, 17] of DDR3 to reduce or increase the effective memory B/W and latency (see Table 9 for configurations used). We assume the highest B/W (11.4 GB/s) as NearMem and, ∼10x slowdown (1.8 GB/s) of DRAM as FarMem.

Impact of memory bandwidth: Figure 2 shows the application slowdown factor as a function of reduced memory bandwidth, and the table in the Figure 2 shows the memory intensity of the applications in cache misses per 1K instructions (MPKI). As it can be observed, for memory, CPU, storage intensive application like GraphChi, XStream, and Metis show a signifi-



	Graphchi	Xstream	Metis	LevelDB	Redis
MPKI	27.4	24.8	9.61	4.7	20.1

Figure 2: Memory B/W impact and MPKI



Capacity vs. Miss Ratio	Graphchi	Xstream	Metis	LevelDB	Redis
1/2	0.3	0.3	0.6	0.0	0.2
1/4	0.7	0.8	0.7	0.0	0.5
1/8	0.8	0.9	0.8	0.0	0.7

Figure 3: Miss Ratio vs Capacity

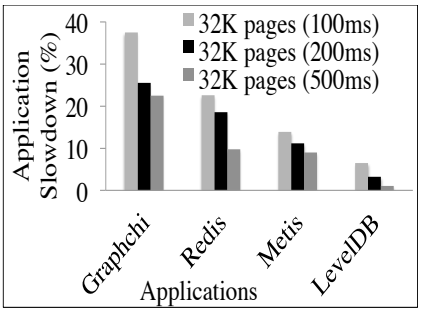


Figure 4: Hot page tracking overhead

cant slowdown with B/W reduction, as expected, given their high MPKI values. Redis shows up to 2x throughput reduction even with a 10GB Twitter dataset [19], and the impact can be higher for larger datasets. I/O-intensive LevelDB shows up to 70% impact for the slowest memory. Interestingly, the Nginx webserver shows less than 30% slowdown even with 10x B/W reduction. *These results show that applications exhibit high variability in their sensitivity to memory speeds, demonstrating the importance of software methods controlling heterogeneous memories to dynamically adapt the memory management actions to the workload characteristics.*

Impact of memory capacity: Figure 3 analyzes the impact of capacity of NearMem. The y-axis shows the ratio of NearMem to the DRAM capacity (4 GB in this analysis) and the table in the Figure 3 shows the NearMem allocation miss-ratio [45] for 1/2, 1/4, and 1/8 capacity ratio. The miss ratio is defined as NearMem page allocation misses to total page allocation for a workload. Because guest-VMs lack the directed on-demand allocation capability to a specific memory type, we use just the OS without virtualization for analysis by binding applications to NearMem NUMA node, and their capacity is varied at the boot-time with "memmap" [6].We use all applications in Table 1 except Nginx due to limited B/W impact. As the graph shows, with 1/2 (2GB) NearMem capacity ratio, most applications except XStream show less than 30% slowdown due to significantly lower miss ratio, which indicates that (1) either the actual number of allocated and used pages is less than NearMem capacity, or (2) applications allocate and release memory frequently. Reducing the capacity to 1/4, XStream, GraphChi shows 52%, 129% performance reduction, and the reduction rate is lesser for 1/8 capacity. This is because, GraphChi, and Xstream are capacity and I/O buffer cache intensive [29]. Current OS prioritize heap allocations, only use freely available pages for the buffer cache. At 1/4 and 1/8 capacity, buffer cache NearMem allocation misses increases, and I/O pages use FarMem pages. For in-memory Redis, and memory intensive Metis, the slowdown is consistent with reduction in capacity and miss ratio. For transactional I/O-intensive LevelDB, the transactions are short with buffer cache used mainly for reads, hence the performance reduction is less than 15% up to 1/32 ratio and increases to 61% beyond that. *Key takeaways are that, several large scale applications frequently allocate/release memory. Providing fast on-demand allocation/reclamation for capacity-limited NearMem, and prioritizing short-lived buffer cache allocations to NearMem for I/O data-intensive applications can significantly increase application performance.*

2.2. State of the art management techniques

Because of limited NearMem capacity, and higher memory footprint of large scale application, recent research such as [32], BATMAN [13], X. Dong et al. [16], CAMEO [12] have discussed purely H/W-based [13, 16], hybrid S/W-H/W [32], and purely S/W-based management [22] methods. Most of these mechanisms focus in detecting critical "hot pages" and migrating such hot pages to NearMem. The earliest such mechanism for DRAM and disk swap was proposed in the seminal work by Denning [15]. It works by setting the “use” bit to ON for each page-table entry (PTE) when its corresponding page is referenced in an interval of time. This approach is applicable for today’s software-based working set detection too. But, compared to the H/W-based approach (e.g., at the cache controller), the OS has to use a coarse-grained reuse information, as discussed by Meshwani et al. [32]. Further OS-based tracking and data movement require several overheads, including repeated scan of the page-tables , setting and resetting the reference bit in the page table, as well as data movement-related overheads due to page allocation, page table updates, TLB flushes and address space locking. Solutions such as BATMAN [13], [16], [32] propose specialized hardware for page monitoring and migration which vary from memory controller-based to TLB-based tracking. We discuss the key differences with VMM and OS-based HeteroMem in Section 7 A more closely related virtualization-based research, Heterovisor, by Gupta et al. [22] proposes a application and guest OS transparent hot-page working set detection mechanism for managing memory heterogeneity exclusively at the VMM. *While VMM-level management of heterogeneous memory is possible. The question we ask is whether it is sufficient in extracting adequate performance benefits from the underlying memory resources.*

2.3. Drawbacks of VMM-exclusive Management

In a VMM-exclusive approach, the VMM has control over the Guest-VM resources, and takes the responsibility of tracking

resource usage, tracks the frequently accessed FarMemory pages and migrates them to NearMem either by new allocation or an eviction of least frequently used NearMem page. Further, using the VMM to detect and manage heterogeneity transparently, without adding guest OSs or application complexity is an appealing option. However, the transparent approach has several drawbacks which we discuss next.

Coarse VMM-level vs. fine-grain OS-level memory representation. The VMM sees the entire guest as one application, and its memory management data structures are designed at an different abstraction level compared to the guest-OS. Hence, the VMM has limited information about the application. However, the guest-OS maintains a fine-grained information about the application. For example, its virtual memory data structures broadly classifies pages into user- and kernel-level, and maintains page flags that indicate the purpose of page allocation – e.g., anonymous, I/O, cache, and DMA pages [21], as well as history of the page usage (active, inactive, swapped) [26, 21] using the active and inactive list data structures. This information can be used to reduce the data movement and migration overhead and improve management efficiency. In contrast, lack of such information in a VMM leads to inefficiencies: For example, migrating a shortly lived/used I/O page cache based on hotness information by the VMM can only add to cost of page movement, rather than providing actual benefit to the application.

Reactive and not proactive. Coarse grain information about the application forces the VMM-exclusive management to always rely on reactive strategies that track page usage and perform migration across memories. Given the overheads associated with these operations (i.e., page allocation, page table updates, TLB flushes, address space locking, and memory bandwidth use during data movement) [3], reactive strategies are beneficial only when migration gains dominate the overheads. To reduce or avoid such overheads, it is important to use proactive strategies that use the guest-OS information to allocate pages right the first time (on-demand), and use complementary reactive strategies only when active memory footprint of an application exceeds the NearMem capacity.

Scope of management. With increasing application memory use, and applications/VMs in a host, the overheads of VMM-exclusive monitoring and management increases. Further, it fails to exploit the rich information and hints from higher layers (guest OS or application) to restrict the memory monitoring and management to specific areas of interest that can significantly improve management efficiency. For an example, a hint from Guest OS about application address space can avoid hotness monitoring for all processes in a VM.

2.4. Impact of VMM-Exclusive approach

To understand the magnitude of the previously discussed issues of VMM-exclusive management, we evaluate a VMM-based working set detection – HeteroVisor, proposed by Lee et al [30] and Gupta et al. [22]. To reduce the overhead of page-table scan overheads at the VMM-level, HeteroVisor maintains a 32-bit history bitmap for each page, and a bit is set after every reference. HetroVisor uses a reverse mapping [25] to walk through pages, an approach available in the Linux-OS known as "rmap_walk()" but, not in the VMM such as Xen, and scans 16K (or configurable count) of pages every 100ms. New pages are added to the list in the front, and scans happen for the tail to quickly account for frequently accessed pages. Figure 4 shows the impact of just detecting the hot-pages (without any migration) for different applications shown on the x-axis, and the y-axis indicates the application performance degradation for different hot-page scan frequency. The hotness tracking for 32K pages in a 100ms frequency adds up to 37% overhead for memory-intensive applications like GraphChi, Metis, and Redis. The impact is relatively low for I/O-intensive LevelDB. Reducing page scanning interval to 500ms reduces the overhead by 5%-10% relative to 32K pages. The overheads, increase with increasing applications or working set size. *The analysis shows that even when using a transparent but optimized hot-page scan can add significant application overhead. This motivates the need for active guidance from the guest-OS to the VMM about regions of interest.*

2.5. Need for Coordinated HeteroMem Mgmt

To address overheads of VMM-exclusive management, we argue that, a coordinated guest-VMM management is required, that enables memory heterogeneity awareness to guest-OSs, and combines the power of guest-OS information with VMM's control over hardware resources for efficient management. However, current virtualization designs do not provide capabilities such as (1) on-demand allocation and management for specific memory types – current memory elasticity mechanisms such as ballooning are VMM controlled (VMM inflates or shrinks a Guests memory size) [44]. They are designed for homogeneous memory, as they do not allows OS to request specific memory type or manage them differently after allocation, and (2) infrastructure for coordinated management – lack of OS-level heterogeneity awareness prevents VMM from using OS-level information about application in monitoring and managing memory efficiently. With HeteroMem, we address these gaps, as described next.

3. HeteroMem Design Principles

We next discuss the key design principles that include OS-level heterogeneity support and the VMM-level extensions required for efficient VMM-Guest coordinated management.

1. HeteroMem Management in Guest OS. The design of HeteroMem is based on the principle that for achieving optimal benefits the guest OS should be aware of memory heterogeneity. The guest should provide specialized management of these resources while also retaining most of the functionality of the existing virtual memory design. In doing so, HeteroMem not only is able to exploit the benefits of existing guest OS

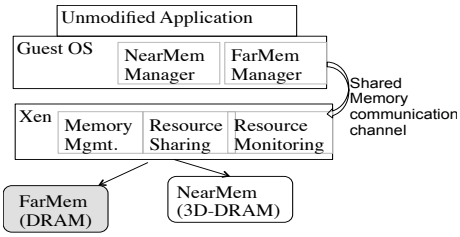


Figure 5: HeteroMem OS and VMM High-level Design

information about memory use, but also achieves a design consistent with the principle of thin VMMs [18].

2. On-demand allocation and management. HeteroMem provides guest-OSs with an on-demand allocation capability from memory types. The term "on-demand" refers to the wide range of allocation such as page-fault on first touch for anonymous and file system memory, or a OS request for an I/O buffer cache page. The memory types are treated as a guest-OS aware NUMA node. Further, for achieving application transparency, OS-level information is used to make 'right' allocations the first time, thus reducing the need for migrations. HeteroMem provides such capability by careful extension of homogeneous virtual memory, that does not combine FarMem and NearMem pages, and provide fast, guaranteed allocation if such page types are available and free.

3. VMM-Guest support for coordinated management. The VMM has direct control over system hardware resources and provides a holistic view of system resource information, whereas the guest OS has fine-grained information about application running over it. Hence, delegating all hardware resource tracking functionality to the VMM, and using the resource tracking information to take actions at the guest OS-level improves the management efficiency. The migrations are done in the guest-OS level to use its information about application and customize and make migrations effective for improving application performance.

4. Fast communication with shared memory. Coordinated VMM-guest management requires frequent communication (tens of milliseconds) of significant information between the guest VMs and the VMM for memory tracking information (hotness) from VMM, hardware performance counter events, or passing hints from the guest-OS to control the threshold of hot page scans by the VMM. In virtualized environments, communication happens with hypercalls, and the overheads associated with hypercalls have been shown to lead to pollution of up to 2/3 of the L1 cache and TLB [42], and significantly jeopardizing performance. For this reason, HeteroMem's design avoids hypercalls and incorporates shared memory-based interface among guest OSs and the VMM.

5. Application transparency. HeteroMem is an application-transparent approach, and only requires guest-OS- and VMM-awareness for memory heterogeneity, thus avoiding the need for new programming models or additional application development complexities. Heterogeneity can be selectively enabled using a application independent shared library to initialize OS HeteroMem support. Further, using a NUMA node-based abstraction provides generic application support for existing NUMA libraries [2].

4. Design and Implementation

We next discuss the design and implementation of HeteroMem, illustrated in Figure 6, and based on the principles summarized in the previous section.

4.1. HeteroMem OS support

Heterogeneous memory as NUMA nodes. To support heterogeneous memory allocation and management at the guest-level, HeteroMem leverages existing OS-level abstractions and mechanisms that deal with diverse memory properties, and represents different memories (FarMem and NearMem) as OS memory nodes. This is similarly to what is currently used for memory nodes in NUMA platforms, and also provides flexible support for other future memory technologies. Information regarding available nodes and their initial capacity is exported by the VMM during guest initialization. While HeteroMem is not dependent on application level hints, however, using NUMA node based design provides a generic support for applications directed allocation too [2].

Guest-OS interface for on-demand allocation. Current guest-OS interfaces are designed for homogeneous memory systems, allowing guest OS to request/release specific number of pages to/from the VMM (e.g., balloon driver interfaces), without ability to request allocation to specific memory type. To enable on-demand allocation, we introduce a HeteroMem on-demand allocation driver (see Figure 6) that requests the VMM to increase/decrease reservation of a specific memory type along with fallback strategy which we discuss shortly.

HeteroMem page management. Unlike homogeneous NUMA (DRAM) support, for supporting heterogeneous memory several virtual memory extensions are required. HeteroMem extensions strive to retain most functionality of existing virtual memory management implementations, and provide specializations only when required. First, new flags NEAR_MEM, DRAM (or NVM) are added to page structure. These flags help classify pages from different memories for providing specialized allocation, reclamation and migration. Using the node-based design retains the existing zones (HIGHMEM_ZONES, NORMAL_ZONES, DMA_ZONES). Further, in current 64-bit architecture, the HIGHMEM_ZONES are empty, which we utilize for NearMem. Using the node-based design avoids the complexity of a new zone (as in [43, 47]), and increase page management complexity.

NearMem Per-CPU free list: For allocations, OSs such as Linux use per-CPU free page lists as a fast path for allocation and reclamation, reducing the use of the 'Buddy' allocator. However, currently this is limited to homogeneous memory only. We extend the implementation by replacing the per-CPU list with an array of per-CPU list with array size equivalent to

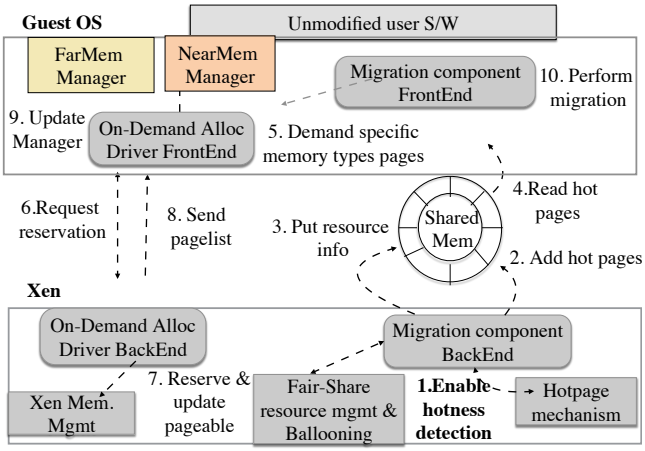


Figure 6: HeteroMem Architecture shows the VMM-level hotness tracking and Guest-OS migration steps

memory types. Since NearMem capacity is limited, almost all NearMem allocation/reclamation happen from the free lists.

4.2. Coordinated VMM-Guest management

Next, we discuss the design and implementation of the VMM-guest OS coordination. HeteroMem's coordinated management design follows a split front-end and back-end model, where the functionality required for management is split across the guest and the VMM, as follows.

VMM support for on-demand allocations. Enabling guest-OSs to demand reservation and allocation of specific type of memory requires additional VMM-level interfaces and fall-back policies. In HeteroMem implementation, we add a new hypercall to the Xen memory management, that allows guest to specify the amount and type of requested memory. The supported policy is to honour guest requests as long as the requested memory is available (strict allocation), or fall back to other memory type, if that is not the case. The VMM remains responsible for memory allocations and page table updates.

VMM-level hotness tracking. To deal with capacity limitations of NearMem, the hotness detection mechanism provides the capability to detect the active working set that needs to be migrated. Section 2 discussed the reverse mapping-based hotness detection mechanism and its benefits and software overheads. Given its highest privilege-level, the VMM is delegated with the responsibility to perform the page-level tracking (hotness tracking) to identify frequently accessed pages.

Guest-OS-controlled migration. The hotness information is exported to the guest OS, and actual migrations are controlled by the guest, for following reasons. (1) *Page state*: Before a page is migrated in the OS from a FarMem to a NearMem node, the OS performs several checks and validation of the source and destination pages, and failure to do such checks can result in application and OS errors [32]. These checks include whether (i) a page is a valid at the time of migration, and mapped to a virtual address space, (ii) the page is not a dirty file system page with uncommitted data, (iii) the page is not reserved by the OS or a DMA-based device, and finally (iv) the page is not linearly mapped to the kernel or DMA space. Unlike the VMM, the OS virtual memory and page structures have all these information readily available. Therefore, the guest-OS controlled page migrations avoids errors. (2) *Scalability via adaptive migration*: With increasing application working set size, and the VMs on single host, the number of pages for migration also can increase. Not all migrations are useful, and the limited information about current page state of an application can result in high monitoring and movement overhead. However, the guest-OS, with fine-grained information about such application page state, can selectively customize, or enable/disable the migrations, and reduce overheads. We discuss such mechanisms in Section 5.

Fast Communication with shared memory. VMM-guest coordinated management requires significant information transfer which includes page hotness information (see Figure 6), hardware counters, NearMem and FarMem resource information, and OS-related hints (discussed in Section 5) for guiding VMM to track regions of interest. To enable this, in HeteroMem system, shared memory pages are reserved during boot with a default size of 128 pages, but can be dynamically extended or coalesced, based on the guest OS needs. The communication data is record structure (e.g., entire hotness information is a record), and a record consists of information type header, length, with first 8-bytes reserved for shared memory synchronization.

4.3. VMM-level resource management:

For, Inter-VM sharing of NearMem, HeteroMem extends the Xen ballooning [44] for reservation and over-committing. The VMs specify their NearMem minimum and maximum size (VM can allocate its maximum size when memory is not over-committed) at boot time – a capability that cloud providers can directly tie a cost model, which is beyond the scope of this paper. The VMM's memory management initializes a minimum, and reserves a maximum NearMem node for the guest similar to DRAM [44]. The initialized minimum size guarantees fairness for guest based on its NearMem priority (or cost model). When a guest runs out its minimum NearMem, the HeteroMem's ballooning driver increases the share by either requesting the VMM to allocate, or activating the ballooning in other VMs to release free NearMem pages. A subtle difference with the current DRAM-based ballooning is that, the HeteroMem's guest-OS balloon drivers use NearMem-LRU mechanism (discussed later in Section 5) by migrating inactive NearMem pages to FarMem, and when not sufficient, swapping them to disk (or to NVMs in the future).

5. Efficient Management

We next present the management policies that focus on (1) reducing NearMem miss ratio, increase NearMem reuse metrics (discussed in Section 2 with on-demand allocation, and (2) reducing migration overheads and increase migration gains –

```
for Each page allocation request do
    Try.OnDemand:
    if Request is heappage then
        Allocate heap.NearMem
        Set heap page flags and return
    end
    else if Request is IO.page then
        Allocate IOpage.NearMem
        Set IO page flags and return
    end
    if Page allocation failed then
        if First OnDemand NearMem Miss then
            Activate VMM Hotness tracking
        end
        Find NearMem Inactive pages
        if Found NearMem Inactive pages then
            Migrate NearMem pages to FarMem
            Release NearMem
            Try.OnDemand again
        end
        else
            Get hot pages from VMM
            Find NearMem.LRU pages & swap with hot pages
        end
    end
end
```

Algorithm 1: HeteroMem page allocation & migration steps

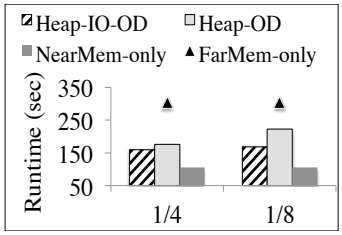


Figure 7: GraphChi: Heap-OD and Heap-IO-OD gains

Batch size	Page migration time	'rmap' walk time
8K	25.5µs	43.21µs
64K	15.7µs	26.32µs
128K	11.12µs	10.25µs

Figure 8: Avg. OS page migration & validation cost

when the on-demand allocation is not sufficient for applications with memory footprint larger than NearMem capacity, frequently used FarMemory pages are migrated to NearMem, and least frequently used NearMem pages to FarMemory either by OS-level management, and/or by using VMM-level hotness tracking information. We term the performance improvement from such migrations as migration gains, which is intuitively the performance difference between a migration enabled approach vs. purely on-demand only approach. Algorithm 1 shows the management algorithm.

5.1. Reducing Overheads via 'On-Demand' Allocations

On-Demand Heap NearMem Allocation (Heap-OD). Before invoking any reactive hotness-based tracking and migrations, using the on-demand allocation capability, the application heap pages are allocated from NearMem, because a significant time is spent accessing the heap memory. More importantly, many large scale applications frequently allocate/release across the application run, which significantly reduces NearMem misses, increases reuse, and reduces migration. Further, NearMem per-CPU page free list cache fasten allocation and release NearMem pages. Figure 7 shows the impact of on-demand heap allocation (Heap-OD) compared to FarMem-only (DRAM-only, worst case), and NearMem-only (best case) for GraphChi. Even with 1/8 NearMem on-demand allocation, GraphChi achieves up to 2x speedup *showing the need for Heap-OD for guest OSs*.

On-Demand I/O Cache NearMem Allocation (IO-OD). While Heap-OD can be beneficial for memory capacity intensive applications, several datacenter-based applications such as databases, graph analytics are I/O (storage and network) intensive. For these I/O intensive applications [29, 40, 1] the buffer cache allocated by the OS play a significant role in improving the I/O bandwidth and application performance. Further, the buffer cache pages are generally active during the I/O phase, and remain inactive or released when free available memory reaches a swap threshold [26]. Enabling on-demand I/O buffer cache allocation (IO-OD) from NearMem improves application speedup for I/O intensive applications. Further, to avoid contention of IO-OD with Heap-OD or hot page migrations, HeteroMem implements an inactive page reclamation (Heap-IO-OD-NMLRU) policy which we discuss shortly in OS-based migration (5.2) discussion. In Figure 7, IO-OD combined with Heap-OD improves GraphChi speedup by 45%. For non-I/O intensive applications, the performance remains unaffected as shown in Section 6.

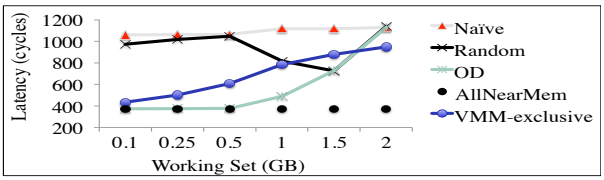
5.2. Reducing Overheads via Guest OS Hints

Proactive Heap-OD and IO-OD are only possible when the active memory footprint of the application fits within the per-guest NearMem. Hence it is important to maximize use of NearMem via necessary cross-memory data migrations. Given the aforementioned overheads of migrations, we next describe the HeteroMem principles that either delay or reduce such hotness tracking by (1) using the existing OS-level monitoring, (2) reducing scope of monitoring using OS-level hints. The outcome of these steps are reduced tracking overhead, increase in NearMem reuse, and avoiding unnecessary migrations.

OS-based migrations (Heap-IO-OD-NMLRU). Existing OS swap management mechanisms use LRU-based algorithms (split LRU in Linux) to classify active and inactive referenced pages. Newly allocated pages are added to a head on the inactive list, and pages referenced only once or never are moved to the end of the inactive list before they are reclaimed, and the promotion happens from the tail to head of active list. The eviction of pages happens only when the memory pressure increases to swap threshold. Specifically for capacity-restricted NearMem, while the classification is insufficient because the total active pages can exceed NearMem, however this can be used as a first step in deciding what pages should be migrated to vs. evicted from NearMem even before depending on VMM-based tracking. Further, unlike current DRAM management, NearMem's LRU immediately migrates and releases the pages once they reach inactive state. As shown in Algorithm 1, when Heap-OD and Heap-IO-OD fails, HeteroMem evicts all inactive NearMem pages to FarMem and releases NearMem pages for subsequent allocations. The VMM-level

Mem. Nodes	2
CPU	Intel Xeon 2.67 GHz
No.of CPU cores	16
FarMem capacity	4 GB per VM (total 16GB)
NearMem capacity	128MB-2GB per VM
FastMem B/W	11.4GB/sec
DRAM B/W	1.8 GB/sec
Storage	Intel-SSD 320 series

Figure 9: Experimental setup



Miss Ratio	Working Set (GB)	0.25	0.5	1	1.5	2
	Random	1.0	1.0	0.92	0.68	0.76
	HeteroMem-OD	0.0	0.0	0.54	0.62	0.76

Figure 10: Memlat and NearMem Miss ratio

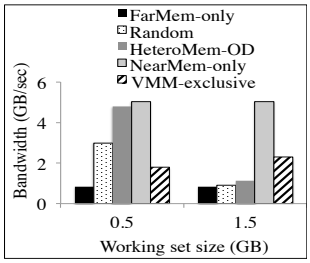


Figure 11: Stream benchmark

aggressive tracking is enabled only if the OS-based evictions are insufficient. This enables, HeteroMem to (1) utilize the active/inactive classification work done by the guest-OS, and (2) delays the VMM-level tracking. As our results show, such OS-level evictions are useful to evict short-lived I/O buffer cache pages and increase NearMem reuse. Further, evictions are done in batches as OS migration and page validation cost decreases with increasing batch as shown in Table 8.

Combining OS-level hints with VMM's hotness tracking: As shown in the Figure 1, hotness tracking is enabled only when Heap-OD, Heap-IO-OD, and Heap-IO-OD-NMLRU are insufficient. To combine the OS-based classification with VMM-based hotness tracking, HeteroMem adds a "HOT" state for pages in the active/inactive list based on the VMM supplied hotness information. The guest OS first tries to evict/migrate "inactive and not-HOT" NearMem pages at the head of the inactive list to slow memory. If no such pages exist, then it falls back to replacing "active, but not-HOT" pages with "active, and HOT" pages to NearMem.

Using OS page classification. To further reduce the scope of VMM-based usage tracking, HeteroMem exploits the virtual memory process regions information by hinting the VMM by restricting the scope of hotness monitoring to contiguous memory regions of a process [21]. Further, all the linearly-mapped ZONE_DMA pages are skipped. To avoid the overhead of tracking I/O buffer cache pages is NearMem, HeteroMem classifies the heap and buffer cache pages in the ZONE_HIGHMEM (NearMem zone), and hints VMM to skip monitoring them, a feature useful for I/O intensive applications. HeteroMem's NearMem LRU mechanism can be used to evict them. Our experiments show improved application performance due to reduction in VMM resource tracking cost.

Architectural hints to reduce migration overhead Prior research [32] have proposed H/W mechanisms to track hotness [32] at the memory controller level can capture both page reuse and access pattern. However, current S/W mechanisms only provide information about page reuse by making the OS to forcefully set the page reference bit, and to quantify the page access pattern behaviour. Applications with high page reuse but low cache misses cannot benefit significantly from page migration, and can only increase application overhead. To address this, HeteroMem combines the architectural hints (application cache use) with reuse (hotness) information for deciding on the interval of hotness tracking and migration. The interval is increased or decreased based on the cache misses by reading the H/W counters in the VMM and using a simple model $Intervalchange factor = (LLC.MISS_i - LLC.MISS_{i-1})/LLC.MISS_{i-1}$, where $i, i-1$ represent the current and previous interval. This information is also exported it the guest-OS via. shared memory.

[0]In Linux, this can be extracted from VMA structures

6. Evaluation

We next present the results from the experimental evaluation of HeteroMem using micro-benchmarks and real-world cloud applications. The results illustrate (1) the importance of OS-level heterogeneity-awareness, (2) the implications and benefits of VMM-Guest coordinated management, and finally, (3) the effectiveness HeteroMem's resource sharing mechanisms.

6.1. Experimental Setup, Methodology and Applications

Table 9 summarizes the experimental test-bed used in our evaluations. Because on-chip stacked-DRAM is not available, as discussed earlier, we emulate memory heterogeneity via memory controller throttling for 10x B/W difference between NearMem and FarMem. The table also shows the different bandwidth speed by throttling memory, and the second and third column show the factor by which the bandwidth and latency are reduced compared to the no-throttling case. Our evaluations use both micro-benchmarks, as well as the cloud applications summarized earlier in Table 1. We use two baselines for all our results, (1) FarMem-only - a FarMem-only DRAM only allocation, as a worst case baseline, and (2) NearMem-only - an ideal case baseline where the NearMem capacity is equivalent to DRAM, and pages are always allocated right to NearMem without migrations.

6.2. Microbench: Access latency and Bandwidth

We first analyze the impact of memory latency and bandwidth using the latency-sensitive pointer chasing 'memlat' benchmark [8], and the memory B/W 'stream' benchmark. Figure 10 presents the average S/W memory latency (in cycles), and the table below shows miss ratio. Figure 11 shows the average B/W for 0.5 and 1.5 working set size. We compare five different policies, (1) FarMem-only (all DRAM), (2) NearMem-only, Random [13] where allocate NearMem pages are added guest-OS DRAM manager without on-demand allocation capability, (4) OD (HeteroMem on-demand), and finally, (5) VMM-exclusive that uses reactive VMM-based

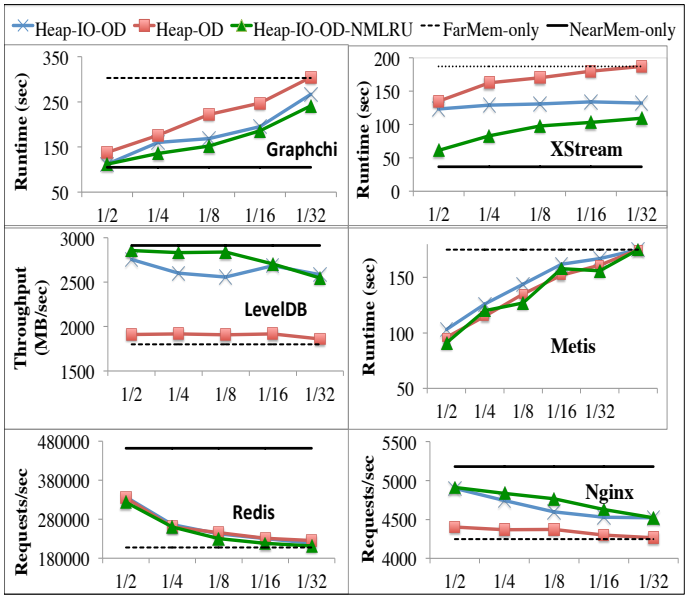


Figure 12: Impact of OS aware design

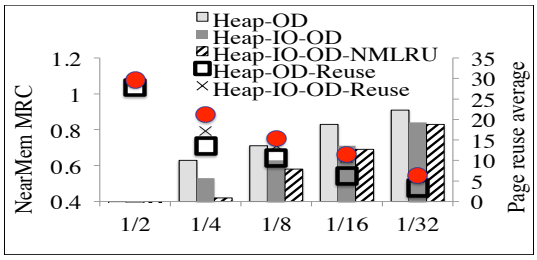


Figure 13: GraphChi NearMem miss ratio & reuse

hotpage detection. We use 0.5GB NearMem capacity and vary the benchmark latency from 128MB-2GB. Also, the benchmark does not release pages after initial allocation.

Observation: Firstly, with the random approach in the Figure 10, the memory latency and bandwidth results show a non-deterministic behavior. In the case of memory latency benchmark, although the working set size is less than the NearMem capacity, the performance is close to worst case baseline, because OS randomly allocates a page, whereas latency decreases at 1GB working set size due to higher proportion of NearMem pages used, and deteriorates for higher capacity. The B/W values also show the same trends. However, for OD, the latency and bandwidth results are more deterministic. Up to 0.5GB working set size, latency is same as the NearMem-only case confirmed by 0 miss ratio, and upon further increase, the latency, and B/W gradually increase and decrease respectively. For the VMM-exclusive approach, while the results are predictable, using a hotness-based NearMem use for a random pointer chasing workload results in higher latency. Beyond the NearMem capacity, VMM-exclusive uses page migration (hence the miss ratio is 0) unlike OD, and results in a lower latency. *These results show the need for directed on-demand allocation capability, and the benefits of hotness detection and migration for larger working set size.*

6.3. Impact of OS-level Awareness

To understand the impact of heterogeneity-aware guest OSs, we first discuss the implications of OS-aware management such as on-demand heap (Heap-OD), Heap-IO-OD, NearMem specific eviction and page migrations (Heap-IO-OD-NMLRU) without enabling VMM-based hotness tracking mechanism. As discussed in Section 5, the page allocations, eviction and data movement are done with information available to guest OS (virtual memory). We use the same set of applications discussed earlier in Section 2.

Figure 12 shows the performance of different applications. Figure 13 shows the miss ratio and page reuse information for GraphChi. The y-axis of each sub-figure indicates the performance metric reported by the application, and the x-axis shows the NearMem capacity ratio relative to FarMem. We compare the performance of (1) FarMem-only - indicated with dotted line, (2) Heap-OD - heap allocation on-demand, (3) Heap-IO-OD - heap plus I/O cache allocation, (4) Heap-IO-OD-NMLRU – Heap-IO-OD plus HeteroMem's NearMem specific LRU-based evictions mechanism, and finally (5) NearMem-only - indicated with dark line.

Heap-OD gains: Firstly, as expected most of the optimizations provide significant benefits over the FarMem-only approach. Next, even with 1/4Th of FarMem capacity, HeteroMem's Heap-OD approach improves GraphChi, Redis, XStream performance by 2.5x, and by 90% and 52% for LevelDB and Metis. Even though the actual memory capacity needs are higher, unlike the microbenchmarks discussed earlier, these applications allocate and release memory continuously, as shown by the NearMem miss ratio, and reuse in Table 13. HeteroMem's guest-aware on-demand NearMem heap allocation provides significant benefits. Next, webserver Nginx has limited impact from capacity and B/W. The difference between NearMem-only, and FarMem-only is 22%. The active heap use is less than 30-50MB [4], hence improves performance by around 4%, and ∼60-90MB is used for storage and network buffer which we discuss next.

Impact of Heap-IO-OD: Next, for I/O buffer cache intensive applications, combining heap and I/O-based on-demand allocations improves speedup over Heap-OD. Even with 1/8 capacity ratio, Heap-IO-OD improves GraphChi, and XStream performance by 23%, and 24% respectively. For XStream, the I/O files are mapped to memory and used for compute also. For the transactional LevelDB, only writes are synced and flushed, whereas reads are mapped to memory, resulting in 28% and 36% improvements for Heap-IO-OD over Heap-OD. For Nginx, by place the network (and filesystem) buffers in NearMem, the gains improved from 2% with Heap-OD to 11.4% with Heap-IO-OD. Finally, in-memory application like Redis, and memory-intensive Metis show no gains. For Metis, the inactive but unreclaimed I/O pages from input caused some contention (∼ 6% slowdown) with Heap-OD, which is addressed by Heap-IO-OD-NMLRU discussed next.

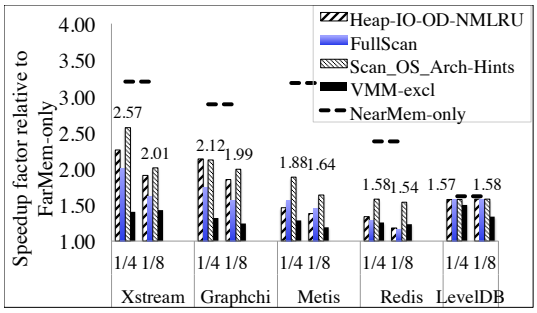


Figure 14: Impact of Coordinated Mgmt

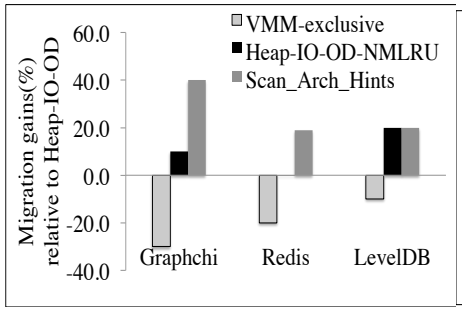


Figure 15: Migration gains (%)

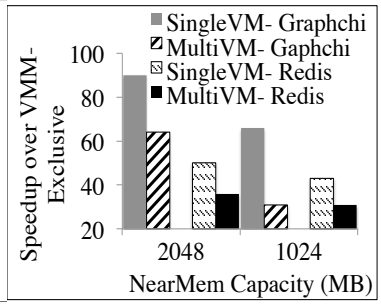


Figure 16: NearMem sharing

Heap-IO-OD-NMLRU: This mechanism finds inactive I/O and heap pages marked by the OS, migrates them to FarMemory, thereby increasing free NearMem pages for subsequent allocations. At 1/4 and 1/8 NearMem capacity for GraphChi, Heap-IO-OD-NMLRU provides ∼12% improvement over Heap-IO-OD and 32% over Heap-OD. For larger (1/2) or smaller (1/32) capacity, Heap-IO-OD-NMLRU provides no or minimal benefits, either because of low miss ratio (at 1/2 capacity), or the active working set is much larger than NearMem capacity (at 1/16 and 1/32). We observed 16% improvement for LevelDB, specifically for read operations. For Metis, by evicting inactive pages, Heap-IO-OD-NMLRU addresses drawbacks of Heap-IO-OD, but the gains are limited$\sim 3-4\%$.

6.4. Impact of Coordinated Management

We next evaluate the impact of HeteroMem's VMM-guest coordinated management using the same applications (except Nginx with low B/W and capacity impact) for 1/4 and 1/8 NearMem capacity configurations. Figure 14 shows the results, and the y-axis represents the speedup relative to the FarMem-only (DRAM-only baseline), and shaded lines represent the the best case NearMem-only approach. The following cases are compared, (1) VMM-exclusive: uses hotness tracking and migration mechanism, without guest-OS awareness, 16K pages of the entire guest-VM is scanned every 100ms (interval based on empirical evaluation, and maximum gains). (2) Heap-IO-OD-NMLRU: as discussed in Figure 12, and note that Heap-IO-OD-NMLRU performs OS-level active/inactive migrations without VMM's hotness scan. (3) FullScan: Heap-IO-OD-NMLRU, and VMM's hotness scan based migrations are enabled. Hotness information is collected for the entire guest's memory. (4) Scan_OS-Arch-hints: represents an optimized FullScan that uses hints from the guest-OS via. shared memory to scan (a) only the application address space instead of the entire guest-VM memory, (b) pages in the OS active list, and (c) combines it with architectural hints by dynamically modifying hotness tracking interval – shorter interval for memory-intensive phase (high LLC misses), and longer when LLC misses reduces (see Section 5), and the interval varied between 50ms to 1sec. The numerical values in the graph correspond to improvement factor for the Scan_OS-Arch-hints case. Figure 15 shows the gain (in y-axis) for Heap-IO-OD-NMLRU, Scan_OS_ARCH-Hints, and VMM-Exclusive approach just from page migrations relative to Heap-IO-OD mechanism with on-demand allocations. For brevity, the graph shows values for three representative applications.

VMM-exclusive vs. Heap-IO-OD-NMLRU: Firstly, as expected, the migration intensive VMM-exclusive approach performs poorly for almost all the applications. For resource intensive GraphChi and XStream, VMM-exclusive's uninformed migration of short-lived I/O pages adds overheads without gains from moving them to faster memory. Figure 15 shows the negative speedup relative to Heap-IO-OD. We also observed ∼40-45% lesser page reuse for Graphchi compared to Heap-IO-OD-NMLRU. For memory-intensive Metis, VMM-exclusive's performance is comparable with Heap-IO-OD-NMLRU, as most pages are heap-based, with high page reuse, and sequential access (MPKI $\sim$ 9.61). In contrast, for Redis with random access (MPKI $\sim$ 20.1), VMM-exclusive suffers from inefficient migrations combined with frequent scan interruptions resulting in 54% lower throughput compared to Heap-IO-OD-NMLRU. Next, for LevelDB, as discussed earlier in Figure 12 (OS-aware measurements), Heap-IO-OD-LRU (and Heap-IO-OD) achieves almost NearMem-only baseline throughput. These results shows that *VMM-exclusive mechanism can add significant overheads for applications with high I/O or random data access patterns.*

FullScan: This approach combines Heap-IO-OD-NMLRU and VMM's hotness tracking. However it tracks the entire Guest-VM's memory for hotness. As a result, resource intensive GraphChi, and XStream experience up to ∼20% overhead even compared to Heap-IO-OD-NMLRU. FullScan for Redis shows similar problems to that of VMM-exclusive, and further reduces Heap-IO-OD-NMLRU's on-demand allocation benefits.However, for Metis, the benefits from migrations is higher compared to unguided hotness tracking, improving the performance by 18% relative to Heap-IO-OD-NMLRU, and showing the benefits of coordinated management.

Scan_OS-Arch-hints: This mechanism outperforms other approaches by combining the benefits of OS-aware on-demand allocations, reducing scan overheads with OS-guided hints, and LLC hints. GraphChi provides ∼90% gains over VMM-exclusive, 16% over Heap-IO-OD-NMLRU. For Metis, the gains are up to 2x over VMM-exclusive, and 29% compared to FullScan. For GraphChi, the scan interval increased up to 700ms during the I/O phase relative to 50ms during the memory-intensive phase. For Redis, with high MPKI, the benefits are restricted to 12% over Heap-IO-OD-NMLRU.

MultiVM NearMem Sharing: Figure 16 evaluates the effectiveness of HeteroMem in sharing heterogeneous memory by extending Xen's fair-share ballooning (as discussed in Section 4) to NearMem also. We use two VM's, one running Graphchi, and the other running Redis. The x-axis shows the total system NearMem capacity. The bars SingleVM-Graphchi and -Redis indicate the gains (y-axis is in %) of Scan_OS-Arch-hints over VMM-exclusive approach when running alone. In the MultiVM case, NearMem capacity is equally shared between co-running Graphchi and Redis VMs. The VM boot configurations for NearMem are, $Min.NearMem = totalsystemNearMem/2$, and $Max.NearMem = totalsystemNearMem$. The applications start at the same time, however, the I/O and memory-intensive Graphchi exhausts its Min.NearMem quota, and requests the VMM to increase its reservation. For Redis, the increase is gradual over time, hence the VMM and HeteroMem balloon driver are able to reclaim unused NearMem from Redis (in 128 MB batches) initially. However, as Redis NearMem usage increases, to satisfy the Min.NearMem for Redis, HeteroMem's balloon driver in Graphchi starts evicting inactive pages, followed by I/O buffer cache, and finally swapping pages to disk. As observed, in the MultiVM case , For GraphChi with 1024MB Min.NearMem , most evictions are I/O and inactive heap pages resulting in a $\sim 28\%$ slowdown relative to the 2GB single VM case. However, for 512MB case, lack of NearMemory results in significant swapping resulting in 40% slowdown. For Redis, with progressive NearMem demand, the slowdown rate is lower. While HeteroMem's fair-share mechanism provides up to 60% benefits over the VMM-exclusive case, a more optimal multi-resource (NearMem, FarMem, I/O cache) sharing mechanism such as [20] can increase benefits.

Summary: Our results show that (1) OS-level on-demand allocation and management methods can be leveraged to improve the use of heterogeneous memory resources reducing migrations, (2) the overheads of expensive S/W-based hotness tracking can be reduced using OS and architectural hints. Finally (3) combining VMM and OS-aware management methods by exposing VMM-level information to OSs leads to additional gains in application performance and management efficiency.

7. Related Work

Stacked-DRAM (NearMem) Heterogeneity: A large body of research has focussed on the use of utilizing stacked-DRAM as a large last level cache [48, 24, 34, 31, 12]. We discussed the issues of using them as last-level cache, and recent proposals such as [13, 16, 10, 23, 32, 33] that propose using them as high B/W DRAM. BATMAN [13] proposed a H/W-based data placement methods focussed in increasing the cumulative DRAM and Stacked-DRAM B/W. Further, BATMAN uses a randomized partitioning model to partition application access between NearMem and FarMem. X.Dong et al. [16] propose methods to optimize data migration using a specialized H/W address translation for mapping the NearMem slots with with page physical address, and modify the memory controller to achieve this. Meswani et al. [32] also make use of a modified TLB to find hot pages. HeteroMem's proposals are S/W-, OS-, and VMM-based, and the H/W optimizations can further complement its benefits.

OS management of heterogeneity: Next, regarding OS-level management, S.Phadke et al.[33] studied the need for memory heterogeneity with multicore system, and proposed an offline algorithm that classifies applications into latency sensitive (pointer chasing), bandwidth sensitive (graphics), or CPU intensive application, and use the analysis to guide OS-level page allocation. However, while such hints can be useful, HeteroMem does not depend on availability of profiles. As discussed earlier, while [32] also discuss some of the on-touch page allocation and migration mechanism, HeteroMem (1) analyses memory, CPU and storage intensive application, (2) proposes a design to manage heterogeneity in virtualized and non-virtualized environments, and more importantly (3) uses the breadth of information from existing OS to make decisions about on-demand allocation, and (4) provides additional data structures, mechanisms and policies that enable Guest-VMM coordinated management. Other research, such as [37], and commercial products [9] provide NUMA support for VMs. While such support complements HeteroMem design, for heterogeneous, capacity constrained memory, DRAM-based management is not sufficient [32], and requires heterogeneity aware active management and migration.

Storage class memory (SCM): Prior work has considered heterogeneity-aware approach to manage storage-class memory [28, 17, 43]. Common technique in these solutions uses application-level libraries that expose heterogeneity by allowing applications to use specialized interfaces. Use of such interfaces/hints may lead to further improvements in HeteroMem, but the focus of the current design is to provide benefits without changes to existing applications. Other virtualization-based research [46, 49] have focused on improving the storage I/O stack, while HeteroMem does not limit the purpose of use.

8. Conclusion

In this paper, we explore the use of heterogeneous memory in virtualized systems, and analyze their impact of different cloud-based applications. To address the drawbacks of VMM-exclusive management that fail to exploit OS-level information, we propose "HeteroMem" that exposes memory heterogeneity to guest OS, with support for on-demand allocation, OS-level management, and support for VMM-Guest coordinated management. We discuss different management principles that strive to reduce data migrations across memories, and increase NearMemory use. Our results show up to 90% benefits over VMM-exclusive approach, and 2x improvement over using FarMemory only. We plan to explore multiple levels of heterogeneity such as NVM in the future.

References

[1] Google LevelDB . http://tinyurl.com/osqd7c8.

[2] Linux libnuma. http://linux.die.net/man/3/numa.

[3] Linux Page Migration. https://www.kernel.org/doc/Documentation/vm/page_migration.

[4] Nginx memory usage. https://www.nginx.com/blog/nginx-websockets-performance/.

[5] NGinx Webserver. http://nginx.org.

[6] Node Capacity memmap. https://www.kernel.org/doc/Documentation/kernel-parameters.txt.

[7] Redis. http://redis.io/.

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

[9] VMWare vNUMA. https://www.vmware.com/files/pdf/techpaper/VMware-vSphere-CPU-Sched-Perf.pdf.

[10] AKIN, B., FRANCHETTI, F., AND HOE, J. C. Data reorganization in memory using 3d-stacked dram. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (New York, NY, USA, 2015), ISCA '15, ACM, pp. 131–143.

[11] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 43–57.

[12] CHOU, C., JALEEL, A., AND QURESHI, M. K. 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* (Washington, DC, USA, 2014), MICRO-47, IEEE Computer Society, pp. 1–12.

[13] CHOU, C.-C., JALEEL, A., AND QURESHI, M. Batman: Maximizing bandwidth utilization for hybrid memory systems. In *Technical Report, TR-CARET-2015-01 (March 9, 2015)* (2015).

[14] COBURN, J., CAULFIELD, A. M., ET AL. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS '11*.

[15] DENNING, P. J. The working set model for program behavior. *Commun. ACM 11*, 5 (May 1968), 323–333.

[16] DONG, X., XIE, Y., MURALIMANOHAR, N., AND JOUPPI, N. P. 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* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–11.

[17] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 15:1–15:15.

[18] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, JR., J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 251–266.

[19] FERDMAN, M., ADILEH, A., KOCBERBER, O., VOLOS, S., ALISAFAEE, M., JEVDJIC, D., KAYNAK, C., POPESCU, A. D., AILAMAKI, A., AND FALSAFI, B. 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* (New York, NY, USA, 2012), ASPLOS '12, ACM, pp. 37–48.

[20] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 323–336.

[21] GORMAN, M. Understanding the Linux® Virtual Memory Manager. http://bit.ly/1n1xIhq.

[22] GUPTA, V., LEE, M., AND SCHWAN, K. 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* (New York, NY, USA, 2015), VEE '15, ACM, pp. 79–92.

[23] GUTIERREZ, A., CIESLAK, M., GIRIDHAR, B., DRESLINSKI, R. G., CEZE, L., AND MUDGE, T. 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* (New York, NY, USA, 2014), ASPLOS '14, ACM, pp. 485–498.

[24] JIANG, X., MADAN, N., ZHAO, L., UPTON, M., IYER, R., MAKINENI, S., NEWELL, D., SOLIHIN, D., AND BALASUBRAMONIAN, R. Chop: Adaptive filter-based dram caching for cmp server platforms. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on* (Jan 2010), pp. 1–12.

[25] JONATHAN, C. Linux object-based reverse-mapping. https://lwn.net/Articles/23732/.

[26] JONATHAN, C. Linux Swapping. https://lwn.net/Articles/495543/.

[27] KANNAN, S., GAVRILOVSKA, A., AND SCHWAN, K. Reducing the cost of persistence for nonvolatile heaps in end user devices. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on* (Feb 2014), pp. 512–523.

[28] KANNAN, S., GAVRILOVSKA, A., AND SCHWAN, K. Reducing the cost of persistence for nonvolatile heaps in end user devices. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on* (Feb 2014), pp. 512–523.

[29] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 31–46.

[30] LEE, M., AND SCHWAN, K. Region scheduling: Efficiently using the cache architectures via page-level affinity. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 451–462.

[31] LOH, G., AND HILL, M. Supporting very large dram caches with compound-access scheduling and missmap. *Micro, IEEE 32*, 3 (May 2012), 70–78.

[32] MESWANI, M., BLAGODUROV, S., ROBERTS, D., SLICE, J., IGNATOWSKI, M., AND LOH, G. 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* (Feb 2015), pp. 126–136.

[33] PHADKE, S., AND NARAYANASAMY, S. Mlp aware heterogeneous memory system. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011* (March 2011), pp. 1–6.

[34] QURESHI, M. K., AND LOH, G. H. 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* (Washington, DC, USA, 2012), MICRO-45, IEEE Computer Society, pp. 235–246.

[35] RAMOS, L. E., GORBATOV, E., AND BIANCHINI, R. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing* (New York, NY, USA, 2011), ICS '11, ACM, pp. 85–95.

[36] RANGER, C., RAGHURAMAN, R., PENMETSA, A., BRADSKI, G., AND KOZYRAKIS, C. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on* (Feb 2007), pp. 13–24.

[37] RAO, D., AND SCHWAN, K. vnuma-mgr: Managing vm memory on numa platforms. In *High Performance Computing (HiPC), 2010 International Conference on* (Dec 2010), pp. 1–10.

[38] RAO, J., WANG, K., ZHOU, X., AND ZHONG XU, C. Optimizing virtual machine scheduling in numa multicore systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on* (Feb 2013), pp. 306–317.

[39] ROSSBACH, C. J., CURREY, J., SILBERSTEIN, M., RAY, B., AND WITCHEL, E. Ptask: Operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 233–248.

[40] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 472–488.

[41] SENGUPTA, D., WANG, Q., VOLOS, H., CHERKASOVA, L., LI, J., MAGALHAES, G., AND SCHWAN, K. A framework for emulating non-volatile memory systemswith different performance characteristics. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (New York, NY, USA, 2015), ICPE '15, ACM, pp. 317–320.

[42] SOARES, L., AND STUMM, M. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–8.

[43] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: lightweight persistent memory. In *ASPLOS '11*.

[44] WALDSPURGER, C. A. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002), 181–194.

[45] WALDSPURGER, C. A., PARK, N., GARTHWAITE, A., AND AHMAD, I. Efficient mrc construction with shards. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2015), FAST'15, USENIX Association, pp. 95–110.

[46] WEISS, Z., SUBRAMANIAN, S., SUNDARARAMAN, S., TALAGALA, N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Anvil: Advanced virtualization for modern non-volatile memory devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2015), FAST'15, USENIX Association, pp. 111–118.

[47] WU, X., AND REDDY, A. L. N. Scmfs: a file system for storage class memory. In *SC '11*.

[48] ZHAO, L., IYER, R., ILLIKKAL, R., AND NEWELL, D. Exploring dram cache architectures for cmp server platforms. In *Computer Design, 2007. ICCD 2007. 25th International Conference on* (Oct 2007), pp. 55–62.

[49] ZHOU, R., AND LI, T. Leveraging phase change memory to achieve efficient virtual machine execution. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2013), VEE '13, ACM, pp. 179–190.

Energy Aware Persistence: Reducing Energy Overheads of Persistent Memory via Efficient and Relaxed Durability

Sudarsun Kannan, Moinuddin Qureshi, Ada Gavrilovska, Karsten Schwan

ABSTRACT

Next generation byte addressable nonvolatile memory (NVM) like PCM is attractive for mobile and other end-user devices as it offers memory scalability as well as fast persistent storage. In such environments, however, NVM's limitations of slow writes and high write energy are magnified for applications that require atomic, consistent, isolated and durable (ACID) updates. Specifically, NVM write/store operations and energy consumption are increased because, for satisfying correctness (ACI), persistent application states must be frequently flushed from intermediate buffers like the processor cache, and to support the durability (D), that state must be logged. This significantly increases NVM access energy cost, and more importantly, it results in additional CPU instructions and energy.

To address the energy overheads of persistence, we develop novel energy-aware persistence (EAP) principles that (1) identify data durability (logging) as the dominant factor in energy increase. Next, we (2) formulate energy-efficient durability techniques that reduce those costs. Examples include flexible logging that switches between performance and energy efficient modes, and a memory management technique that trades capacity for energy. Finally, we propose (3) a relaxed durability (ACI-RD) mechanism for critically low energy conditions, that does not affect application correctness. (4) We evaluate the energy benefits and performance implications of using the EAP principles with realistic applications and benchmarks. Our experimental results demonstrate an up to 2x reduction in CPU and NVM energy usage compared to traditional ACID-based persistence.

1. INTRODUCTION

Industry announcements [1] claim future byte addressable, nonvolatile memory (NVM) technologies like phase change memory (PCM) to have 100x lower access latency compared to that of Flash/SSD devices and to scale to 4-8 times the density of DRAM, without consuming refresh power. The promised outcomes include: larger memory capacities [2], at lower energy usage [3] compared to DRAM, coupled with faster persistent data storage and access than Flash/SSD. Given this, current end-user devices such as smartphones, tablets, and laptops with limited DRAM capacities and slow flash storage [4] are an attractive target for NVM deployment, particularly when using NVM for both high memory capacity (volatile heap) and fast data persistence [3, 2, 5]. Using NVM for memory-based persistence, however, requires systems to provide fail-safe guarantee from application or device failures. For example, when saving an userid (key), and password (value) into a persistent hashtable, a failure after saving the key, but before saving the password can result in undesired result.

To guard against such problems, the concept of ACID - atomic (A), consistent (C), isolation (I), and durable (D) – updates for persistent memory, used with databases, has been well studied [6, 7, 8, 9, 10]. In ACID, 'A' guarantees that either all operations or none completes in a transaction, whereas, 'C' requires each update to convert persistent data from one consistent state to another. For example, C ensures that for a key-value pair their data are updated before the hashtable points to the corresponding key-value pair, else a failure can create dangling hashtable pointers. D ensures data written by a programmer is available after crash, and is achieved by logging. Logging either copies application state to a log before updating in-place (UNDO), or directly writes data to a log, and an the end of a transaction commits the log to the actual location (REDO). Finally, 'I' ensures concurrent updates are invisible to each other realized through race free mechanisms, and synchronization. The ACI of ACID is required for application correctness, i.e., after a failure, the system should be able to detect if a failure occurred, and the point of failure to revert changes after that, whereas D is required for recovery.

For correctness, a consistent and durable metadata about application data is required. To explain the difference between metadata and data, Figure 1 shows a persistent hashtable-based key-value pair insert. The steps include allocating a key-value pair with persistence-aware library allocator, logging the allocator metadata required for memory-based persistence, updating the key-value data, and finally, commit. The commit involves first logging the key-value data (known as record), logging the record header (with a pointer to record, size), followed by pointing the hashtable to the key-value pair, and logging them. After a failure, the library metadata log is used to restore the application's persistent heap, whereas the application metadata (record

headers) restores data to a consistent state before failure. Note that, the energy cost for small fixed size metadata (16 bytes) updates are typically much less compared to data. These complex ACID-related steps performed by the software stack cause undesirable increases in energy consumption, both in terms of NVM write energy and more importantly, the CPU energy.

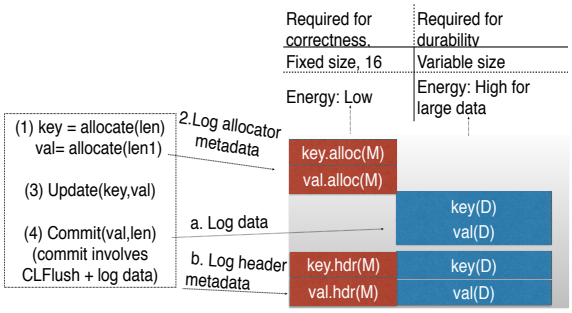


Figure 1: Hashtable data and metadata logging. M, D indicates metadata and data respectively

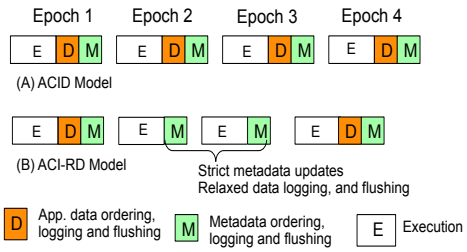


Figure 2: (A) Traditional ACID, (B) ACID-RD: Data logging relaxed for critically low energy Epoch 2,3

To address this, we propose **energy-aware persistence** (EAP) as a basic element of deploying NVM in end-user devices. While prior work has sought to reduce the memory-based persistence ACID performance costs [6, 7, 8, 9] using performance-centric schemes that either optimize transactions for performance or apply optimizations oblivious of the durability and correctness-related persistent components and hence, are unaware of their energy impacts. In contrast, (1) EAP analyzes the energy implications of the correctness and durability components in detail. (2) To the best of our knowledge, these analyses are the first (i) to identify that durability-related costs are the most significant contributor to persistence energy usage, and (ii) to quantitatively demonstrate the significance of the CPU-related vs. solely NVM-related costs in the overall energy overheads of supporting ACID. (3) EAP then leverages these observations to incorporate energy-efficient durability alternatives that trade performance or memory capacity in a controlled manner, to meet the constrained energy budgets of modern end-client devices. When such optimizations are insufficient, (4) under critically-low energy budgets, EAP further improves energy usage by offering a novel relaxed (not delayed) durability model – ACI-RD – that does not affect application correctness, illustrated in Figure 2. The figure provides a timing diagram comparing a traditional ACID vs. our ACI-RD design. In the ACID case, after application execution, both application data and metadata are ordered, flushed and logged, whereas with EAP, when energy is critically low, data durability is relaxed for application data, but not for its metadata, thus maintaining the ACI properties essential for correct operation after failure. EAP operates with an epoch-based execution model, where the target energy-level at the start of each epoch drives the choices of suitable alternatives from (3) and (4) above, which are in turn driven by a dynamic understanding of the durability-related energy of each ACID stack component.

The EAP techniques developed and evaluated in this paper focus on memory-based persistence [8], with energy efficient principles at both application and OS-level. By considering all these elements, and with its dynamic awareness of energy budgets, EAP's energy-aware persistence mechanisms reduce NVM and CPU energy usage by 9% and 31%, respectively, and when combining with ACI-RD, CPU and NVM energy use are reduced by up to 2x.

In summary, this paper makes the following technical contributions:

- *Durability cost:* We identify data durability as the key source of energy bottlenecks for applications with ACID-based persistence (Section 3).
- *Energy-efficient durability:* To reduce durability-related energy overheads, we propose energy efficient principles that trade off performance and capacity for energy when the available budget is low (Section 3).
- *Epoch-based Relaxed Durability (ACI-RD):* For critically low energy budgets, we design a novel ACI-RD model that relaxes data durability based on the energy usage at each epoch, without affecting application correctness (Section 5).
- *Experimental evaluation:* Evaluation with realistic benchmarks and applications on representative end-user platforms demonstrate that EAP can deliver substantial benefits (Section 6).

2. BACKGROUND AND RELATED WORK

Byte addressable NVMs. NVMs such as PCM are byte addressable persistent devices expected to be 100x faster (read-write performance) compared to current SSDs [7, 11, 12]. Compared to DRAM, these devices have higher density scaling as they can store multiple bits per cell with no refresh power, with known limitations imposed by an endurance of a few million writes per cell. These attributes make NVM a suitable candidate for replacing SSDs, but in addition, NVM can also be used as memory, placed in parallel with DRAM, connected via the memory bus. This is because in contrast to SSDs, NVM can be accessed via processor load/stores (read/write), with read speed comparable to that of DRAM, but writes around 10x slower due to high SET times [13], with around 15-40x higher write energy [14, 15].

2.1 NVM Usage Models

Considering the advantages and limitations of byte addressable NVMs, different usage models have been

studied. Early system software and hardware research on NVM [2, 15] has proposed a hierarchical model in which NVM is hidden behind memory, and used as a fast swap space, with a persistence model similar to current SSDs. Although in this model the application/OS abstractions do not change, supporting persistence with ACID guarantees requires data to be flushed and moved across several levels before committing it to NVM. Other research proposed NVM memory-based file system (NVM-Hybrid-FS) by redesigning the filesystem to reduce the I/O software stack cost [12][7][11] . Applications use standard block I/O-based or memory mapped (mmap) interface. When using the filesystem block I/O interface, the redesigned filesystem provides ACID guarantees, however, frequent OS interactions are still required. This limits the benefits of byte addressability. When using the 'mmap' interface, the OS does not have intervene in applications' NVM accesses, and hence, applications are responsible for handling the ACID requirements [11].

To overcome such limitations, researchers have been exploring the hybrid memory model, where NVMs are placed in parallel to DRAM, accessed through memory controllers, and are exposed to application as a persistent extended byte addressable memory [16, 17]. The application interface as well as the OS interfaces are designed for heap-based access. Applications use a load-store interface supported by a persistence-aware library allocator, and transactional support. While this requires changes to applications [18] to use persistent heap-based interfaces unlike the NVM-Hybrid-FS approach, its avoidance of system calls reduces latency and improves throughput, and application ACID properties are managed at user vs. system-level. In this paper, we use NVM the heap(memory)-based persistence model, and analyze applications that access NVM for both persistence and capacity [19, 3].

2.2 Persistence and Overheads

Durability and consistency. NVMs' hardware and software stack must support required levels of consistency and durability across application sessions. To reduce the impact of slow writes, the processor cache can be used to buffer writes and hide write latencies. But in a write-back cache, data can be evicted in any order that affects application correctness after failure. To overcome this, prior efforts have proposed NVM ordering by using write-through caches [16], or with epoch-based cache eviction [7] via memory barriers. Further, durability can be affected during power failures or device crashes, leaving the application in a non-deterministic state, e.g., due to partial updates (note that both data and metadata must be saved consistently). A common approach to deal with this relies on application commits, which in turn trigger frequent cache flush to reduce the possibility of non-recoverable failures. However, to guarantee ACID additional transactional mechanisms such as logging (e.g., undo/redo) are needed. Since flushes and logging are expensive, recent work [6],[5, 8, 20, 9] has (i) proposed optimizations to existing persistence mechanisms in hardware and software [6, 5, 20] with a key goal of reducing NVM access, and (ii) considered relaxed consistency models [8, 9] that do not impact application performance. We categorize such work into two broad categories: (1) persistence efficiency and (2) relaxed persistence, discussed next.

Persistence Efficiency. J. Zhao et al. [6] propose micro-architectural changes that use hardware-based nonvolatile caches to avoid logging costs, by enabling multiversion support for data with in-place updates. Zhao et al. [20] also propose a memory controller redesign in which requests are classified into non-intensive, streaming, random, and persistent, for fairness in handling different request types. Other work such as [5] and [3] explore dual use for persistence and capacity needs. R.S. Liu et al [3] propose a memory controller interface that transparently identifies writes to the persistent NVM region from those to the volatile regions, since the latter do not require persistence support, whereas [5] discuss the cache sharing conflicts when concurrently corunning persistent and non-persistent threads.

Relaxing Persistence. Pelly et al. [8] propose a consistency model that enables reordering of the commit sequence to achieve higher concurrency by asynchronous commit of independent transactions. Such reordering happens only for writes to cache, whereas writes to NVM from cache are still ordered. This constitutes a variation of the epoch-based consistency model that requires hardware-software interaction (changes to H/W) for reordering writes. Y. Lu et al. [9] propose a loose ordering consistency (LOC) protocol comprising of an eager commit protocol, and a speculative commit protocol for relaxing intra- and inter-transaction ordering respectively. Memory log area is divided into blocks of 64 bytes with one metadata header for 7 blocks. Committing log data in block-groups enables, and after a failure, consistency is validated for block groups. LOC assumes the presence of a nonvolatile cache. For traditional database in NVRAM, Pelly et al. [21] propose a group commit protocol that delays logging, and commit for a group of transactions and just updates volatile buffers. It uses existing page-granular updates used in disk-based logging.

EAP vs. Prior Work. EAP differs from the prior work reviewed above [8, 9, 20, 21] in several key ways. (1) EAP identifies and addresses ACID software components that increase energy. (2) Other than NVM support, EAP does not require additional hardware changes. (3) Prior work principally seeks improvements in application performance, a case in point being the lazy/asynchronous, or relaxed atomicity and ordering in [9] and [8]. Using such methods may change when certain actions are taken, but they do not reduce the total CPU instructions being run or NVM accesses being performed, and hence do not directly reduce the persistence-related energy consumption. (4) EAP's relaxed durability models are aimed at memory-based persistence where it is important classify data and metadata for maintaining correctness of heap and application state unlike [8]. Finally, (5) our mechanism's are

driven by the current energy availability, with flexibility to switch between performance, and energy-efficient modes.

3. DECONSTRUCTING THE ENERGY OVERHEADS OF PERSISTENCE

To understand the energy overheads of persistence, we perform detailed analysis of the system's software implementing correctness and durability ACID components. We first formulate a simple energy model that considers both NVM and CPU energy for these components, and use the model to analyze the energy impact of each ACID component. These analysis results are the basis for EAP's subsequent energy-efficient durability principles and ACI-RD design. We focus our analysis to NVM as a memory persistence-based applications and benchmarks use only NVM (instead of a DRAM-NVM hybrid model) with processor cache [19]. While our analysis is for PCM-based NVMs (40x higher write energy than DRAM), the principles for reducing durability energy cost proposed in this paper is applicable for other competing technologies such as STT-RAM (5x-10x higher write energy state than DRAM).

3.1 Component-level Energy Analysis

Several user- and system-level components are involved in satisfying applications' ACID requirements. User-level components include the actual application code and the user-level libraries that provide applications with persistent memory allocation/de-allocation and transaction support. System-level components can be a persistent filesystem or persistent memory management mechanisms [17][5], each maintaining ACID properties for their internal, system-level persistent data and metadata structures.

Model. The total energy consumed on the device will be the sum of energy used by these components, where to satisfy ACID properties, each component should order, flush, fence, and log their states. Energy-aware optimizations, therefore, must address these components and their joint operation. Stated more precisely and focusing on the major contributors of energy consumption – CPU and NVM – the total energy usage of an application is captured by the following equations:

$$\left.\begin{aligned} E_{total} &= E_{APP} + E_{datlog} + E_{metalog} + E_{flush} \\ EA &= E_{datlog} + E_{metalog} + E_{flush} = E_{total} - E_{APP} \\ EA &= EA_{CPU} + EA_{NVM} \end{aligned}\right\} \quad (1)$$

where, E_{APP} denotes energy without ACID, E_{flush} - the energy from cache flush, fence and drain, E_{datlog} - the energy from data logging, $E_{metalog}$ - the energy from metadata logging, and EA - the energy for maintaining ACID guarantees.

Component-level Energy Analysis. We next analyze in more detail the component-level energy consumption for NVM-based applications, with a focus on CPU and NVM energy expenditure.

Hardware basis. We use an x86-based Haswell desktop-based system running the 3.9.0 Linux kernel, with 32 KB L1, 4MB LLC write back cache, Intel 520 120 GB flash memory, and 4GB DDR3 DRAM out of which 2GB is used for NVM. For emulating the slow-reads/writes seen in NVMs, we use hardware counters to delay application load/stores with the delay model described in [22, 11]. For the CPU and NVM energy measurements, we use the RAPL support for CPU and DRAM energy [23], and using the total NVM load/stores (that miss last level cache), we estimate the NVM energy usage [15]. We discuss the methodology for dynamic energy evaluation later in Section 6.

PersistMem apps	Description	Workload
Binary tree [24] (Bi-tree)	Cache & memory inefficient compared to other tree data structures	500K operations with random insert, read, delete operations
B-tree[24]	Persistence-friendly, used in databases, S/W cache etc.	Same as Bi-tree
Hashtable	Used in key-value stores and in-memory caches	Same as Bi-tree
SQLite [25]	Database used extensively in client devices	500K operations of SQLite benchmark [26]
Snappy [27]	Fast compression library used in chrome and other Google products. We ported it with memory-based persistence	1.5 GB directory of image, video, audio, document files
JPEG [28]	Well know image conversion service that we ported with memory-basesd persistence	40K .jpg files converted to .bmp

Table 1: Applications

Applications. Table 1 shows the different client-centric persistence benchmarks similar to [9] and end-user applications with the workload used. For analysis, we only use benchmarks discussed below, and discuss Snappy, and JPEG in Section 6. (1) B-tree is highly persistence-efficient data structure compared to Binary tree, and the hashtable. They are extensively used in databases, large graph, and even in memory management mechanism [18] due to their O(log n) worst case. Each node in a B-tree can have a large number of keys or child nodes, with keys packed in an array that fit well into a cache line, and have smaller depth, and lower cache misses. However, they need to be frequently re-balanced, triggering data movement, new memory allocations and deletions resulting in high persistence overhead. (2) Binary trees with O(n) worst-case [29], are highly persistence- and memory-inefficient. Any update or delete results in flushing and logging the node and its two child pointer cache lines. The tree traversal dominates the cost compared to persistent updates. Next, for (3) Hashtables, used extensively key-value stores and in-memory caches, providing strict ACID support is challenging because, for each hashtable entry with variably sized key-value elements, at least three persistent allocations (hash entry, key, value) should be allocated, and for every update, the hash entry, the key-value information, and the hash structure metadata should be updated. The overhead becomes more pronounced when the allocated entries are insufficient or when there is chaining. Fi-

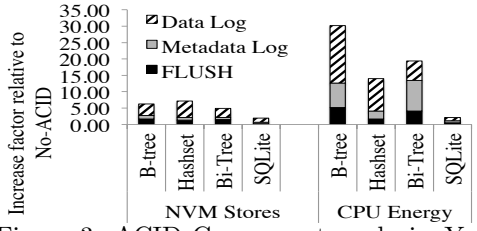


Figure 3: ACID Component analysis, Y-axis: Increase factor relative to No-ACID (baseline)

nally, (4) the SQLite database supports multiple logging mechanisms, including write-ahead logging (WAL), journaling/rollback (sometimes referred to as UNDO), and atomic page commits to avoid logging when hardware provides such features. We next present the component-level energy analysis of these applications. For memory-based persistence APIs, logging, and transactional support for these applications, we extend and optimize the SNIA standard-based persistent memory logging library [18]. For SQLite, we use the existing in-memory database and logging, but with page-based (as opposed to object-based) persistence due to the implementation complexity.

Analysis. Figure 3 presents the impact of providing ACID properties on CPU energy and NVM writes, with (1) FLUSH referring to cache flushing, fencing, and drain operations [30], (2) data log referring to application data durability (logging), and finally, (3) the metadata logging required for durability and application correctness. The x-axis indicates the applications, and the y-axis indicates the increase factor relative to a no-ACID case without any persistence guarantees. As the graphs show, supporting ACID for NVM persistence significantly increases CPU energy (and instructions) and NVM access. More importantly, the durability component (logging) of an application dominates overall CPU energy and NVM write increases, i.e., CPU energy usage just from logging increases by 25x, 12.25x, 10.98x, and 2.68x for B-tree, Hashtable, binary tree, and SQLite. Of the overall logging cost, data logging dominates with 17.55x, 9.86x, 6.8x, and 101% for B-tree, Hashtable, binary tree, and SQLite respectively, with other overheads from metadata updates. The increase in NVM access shows similar trends, but is less pronounced compared to the CPU energy changes. An interesting observation is that metadata logging cost also adds significant cost (up to 5x) for both CPU and NVM writes. This shows that frequent metadata updates not only increase CPU instructions, but also NVM writes from random access. Most of the metadata overheads arise from frequently allocating/freeing persistent memory regions, because after every allocate/free, the persistent metadata of the allocator must be logged for application correctness.

Summary. The results show that supporting fail-safe memory-based persistence can be expensive by several factors compared to a no-ACID case (no guarantees), and that such costs stem mainly from durability operations. Hence, there is a need for energy-efficient and relaxed durability methods under critical energy budgets. Also, for workloads with frequent, persistent allocation/garbage collection operations, it is important to reduce metadata logging costs.

3.2 Deciphering Durability Costs

We next analyze the sources of application data and metadata logging costs. We then use the resulting insights to formulate a set of EAP principles for reducing energy usage. We continue to use the applications employed in the component-level analysis above.

3.2.1 Logging Methods

Prior research on NVM memory-based persistence have used either (1) UNDO (journal) logging [17] or (2) write-ahead logging (WAL) [16]. Also, to exploit byte addressability, memory-based persistence use word or object based logging unlike page-based logging in disk-based systems. We next discuss the energy implications of such logging methods.

UNDO vs. WAL logging for memory-based persistence. The UNDO logging mechanism maintains a journal which resides in persistent storage. With UNDO logging, (1) old data is first copied from the original address to a journal (log) with a record header, (2) then the original address is modified in-place; (3) if a transaction is complete and committed, the log is discarded, else, (4) if it fails or aborts, the log is used to revert updates. Hence, UNDO results in writing twice for every transaction by allowing in-place updates. To reduce the double write overheads, in WAL, all updates are appended to a log, and when the log space runs out, its contents are checkpointed to the original location. WAL maintains a log index to locate and fetch the newest copy of the data in the log before data is committed, whereas, in UNDO, reads are always serviced from the original data location. By always appending to the log, WAL provides sequential writes. Avoiding random writes provides significant benefits for disk-based systems, but gains are limited to memory-based systems, as we later show.

Logging granularity. The NVM logging granularity varies from page-level logging in traditional databases to object- or word-based logging in memory-based persistence applications. While using page-based logs provides easier adaptability for databases to use NVMs instead of disk, there are several drawbacks. As discussed by Pelly et. al [21], first smaller updates to a page must be logged apart from updates to a volatile buffer cache page, and during transaction commit, the original NVM database page must be copied to an UNDO before persisting new updates. However, several recent efforts including [8, 16, 17] have discussed the drawbacks of the multiple step page-based logging which limits the direct byte addressability benefits. To address this, they propose a word or object-based logging, where each word or object (address) is recorded using UNDO [17] or WAL [16] using its NVM address, without considering the page it belongs to. We next discuss energy and performance trade-off for WAL and UNDO using object-based logging.

As discussed by [19, 11], although WAL in memory-based persistence applications can significantly improve

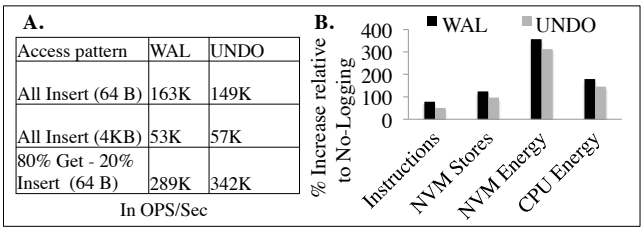


Figure 4: B-tree: WAL, UNDO (A) OPS/sec for individual access pattern,(B) Cumulative access pattern cost relative to No-ACID

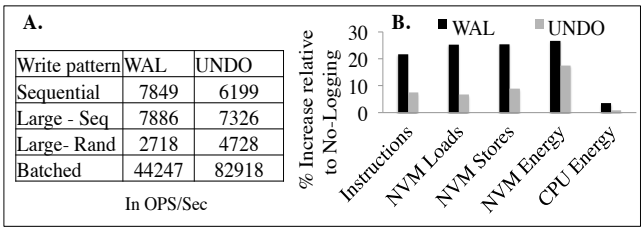


Figure 5: SQLite: WAL, UNDO (A) OPS/sec for individual access pattern, (B) Cumulative write pattern cost relative to No-ACID

performance and concurrency for large multicore systems with write-intensive workload, for read intensive workloads, the improvement is negligible or no improvement at all. This is because servicing read operation requires looking up the WAL index to locate the data, with multiple data versions in the log, which is contrary to the immediate access to the most recent data in UNDO. (2) Also, when updates are large, the fixed size log buffers have to be frequently checkpointed. Checkpoints require parsing the log records sequentially, and copying multiple versions of the same data to the original location. While WAL makes updates faster by only appending to the log, eventually all log entries should be committed, and hence this does not change the total CPU instructions or NVM access. These issues are relevant for page-based logging mechanisms too [31, 32], as used in SQLite. More generally, a key issue in using NVM as heap is that, unlike UNDO, after an update of data, WAL requires that every load/store to same data must be redirected by software to a log, instead of its original location. This not only requires significant application code changes, but also results in additional CPU instructions.

Analysis. To verify the issues, we use the B-tree workload. Because NVML only provides object-based UNDO logging, we add the WAL-logging mechanism. We also use a traditional SQLite workload, and compare UNDO and WAL logging. The system setup is same as discussed earlier. Table 4.A shows the throughput (OPS/sec) for different access patterns, whereas, Figure 4.B shows the increase in CPU instructions, NVM writes, and energy for WAL and UNDO relative to No-ACID (and no logging) for an entire benchmark run with different access patterns. Table 5.A and Figure 5.B show the same numbers for SQLite. As shown, for small, sequential, write intensive workload as in the case of SQLite, and All-insert for B-tree, WAL outperforms UNDO. For read intensive (20% writes, 80% reads) and large value size updates for B-tree UNDO performance is better, and similar results are observed for large updates in SQLite. Importantly, the NVM writes, CPU instructions, and CPU, NVM energy are significantly lower for UNDO relative to WAL in both B-tree and SQLite. *The analysis shows that the performance and energy use of logging methods are highly workload, and implementation specific. Hence right mechanisms should exist to switch between energy and performance modes.*

3.2.2 Metadata Durability Cost

Concerning the metadata persistence overheads, persistent memory allocators can impact the energy overheads for applications that frequently allocate and deallocate data structures (e.g., key-value stores, B-tree), because such allocations, and corresponding data structures should also be logged. Prior work [5] have proposed NVM-write aware allocators to reduce NVM access by placing complex data structures into DRAM, and just using a more efficient allocation log in NVM. However, for small, and frequent allocations, logging and flushing allocator state updates can become expensive, when *metadata/data* ratio increases resulting in more CPU instructions, NVM accesses, and metadata durability energy cost.

4. ENERGY EFFICIENT DURABILITY PRINCIPLES

The energy efficient durability principles formulated in this section are a first step towards energy reduction under strict energy budgets for end client devices.

4.1 Logging flexibility and optimizations

The analysis in the previous section report that although WAL provides lower transaction latency compared to UNDO logging for small updates and write intensive workloads, it leads to increased CPU and NVM energy usage for large update and read intensive workloads, as a result of increase in CPU instructions and NVM accesses.

Logging flexibility. Motivated by these observations, EAP provides *a dynamic logging mechanism able to switch to an energy-aware logging mode (WAL to UNDO).* It is realized in application-transparent way: (1) a persistent application starts in a default WAL mode; (2) after every 'epoch' (fixed time interval), the logging library measures the available energy budget and usage; (3) when energy usage from durability is high, the UNDO mode is enabled for all subsequent transactions. A restriction enforced is that for transactions with a dirty (uncommitted) WAL log that is not yet committed, UNDO logging cannot be used. This is because UNDO requires in place updates, which would result in committing the dirty WAL record first, followed by adding an UNDO log, and hence, cause additional writes.

Library and OS-level optimization. EAP uses the memory-mapped file system support of Intel's PMFS-based OS [11] and the NVML user library for supporting flexible logging. PMFS provides memory-based interface by mapping a region of memory, and using a modi-

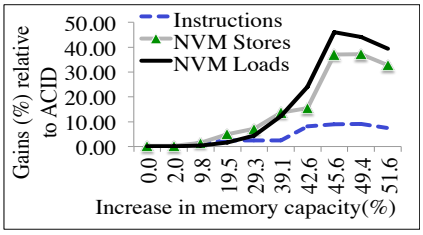


Figure 6: B-tree: Y-axis gains(%) from energy efficient memory mgmt. relative to ACID, x-axis- memory capacity increase(%).

Instructions	NVM Loads	NVM Stores
11.86%	9.28%	10.68%

Table 2: SQLite energy aware memory mgmt. gains relative to ACID, 35% higher memory capacity use

fied 'msync' (memory sync) that internally flushes cache lines (CLFlush) protected by memory barriers, and a store that fences for persisting data. As a first step, we allocate all logs to large 2MB pages in batches of 64MB per allocation instead of regular 4K pages. This is because logs are frequently allocated and deleted, hence using 4K pages increases the OS-level page allocation/deletion and OS-level soft page faults. Further, in an object-based logging approach where each object can reside on multiple physical pages, the current PMFS OS does not check page modifications, and just persists, resulting in redundant writes from developer/user-library's defensive persistence programming. Specifically for metadata, such operations are frequent. To reduce such overhead for logging, EAP extends PMFS with a page-level dirty bit. All pages are write-protected. When a page is accessed for the first time, its dirty bit is set on a fault, with write protection disabled at the OS. When 'msync' is invoked, the dirty bit is reset, and the write protection is enabled. If the page is msync'ed with dirty bit not set, then all persistence operations are neglected ($\sim$ 5.8% pages for B-tree). This reduces both NVM writes and CPU instructions. We discuss more optimizations in Section 5.

4.2 Gain Energy by Trading Capacity

We next discuss the principles for reducing persistent memory management overheads discussed in Section 3 by trading capacity when energy availability is less.

Reduce allocator work. Modern DRAM and persistent memory allocators strive to provide fast allocation and reduce memory fragmentation by maintaining a free pool of memory objects of different sizes (power of 2), and service request by allocating an object from the most optimal pool. When the requests are not aligned with a nearest pool size, smaller objects are merged to match the requested size. When such merging is ineffective, they request the OS (via 'mmap' or 'break' calls) to allocate a batch of pages (64K - 2MB) to refill the free object pools. All such operations consume considerable CPU energy. Further, smaller page allocation batches to the OS result in frequent OS requests (more instructions), whereas larger batches impacts memory capacity due to higher fragmentation. Note that persistent memory allocators not only require pages (allocations) for application, but also for the allocator-based persistent object state management. Generally, objects are maintained in B-tree [18] for O(log n) lookup, and with increasing objects, lookup cost increases. To reduce the energy use (and instructions and NVM accesses), we (1) adaptively use large batched allocations only when the energy budget is constrained, and (2) avoid the complex merge operations that reduce fragmentation unless free available memory is below a threshold (detected using the system swap information), thereby trading off capacity with reduced energy.

Reduce garbage collection overheads. Most garbage collection mechanisms follow a 'mark' (marking an object/page for the collection), and 'sweep' (actual release) mechanism. Garbage collection in end-user devices [33] and server machines [34] has well-understood DRAM costs. Performance and energy overheads are even more significant for NVM, because allocators should persist (and log) deletions, update and persist metadata, and finally release pages by garbage collection, which release pages to the OS internally using expensive 'munmap' calls. Under energy constraints, by trading capacity, the garbage collection can be delayed without affecting correctness. This is done by marking each persistent memory object for deletion, and freeing the objects and their pages in batches only when the available NVM capacity reaches below a threshold. When the capacity threshold is reached, the oldest marked objects (using an inactive object flag) are removed to meet the threshold. We extend the capability at the OS-level too, where garbage collection of a page, and other swap-related page release is delayed by just adding an additional one bit "PG_FREE_DELAY" flag to the existing page flags.

Analysis. To evaluate the above described principles, Figure 6 shows the combined effects of energy efficient allocation and delayed garbage collection, that reduce NVM load-store access and CPU instructions relative to basic ACID-based approach (y-axis in %), and the x-axis show the increased NVM capacity as a trade-off for energy. As expected, with increase in NVM capacity use (so lesser allocator and garbage collection work), the NVM access, and instructions reduce. The benefits from NVM access are higher, because recycling objects require several memset, and memcopy operations, FLUSH, B-tree insert operations (discussed earlier). The reduction in gains at around 45% capacity increases is attributed to, (1) capacity use reaching threshold followed by work done to release memory, (2) the time to lookup or add an persistent object to the B-tree used by allocator increases. We observed similar results for SQLite (see Table 2), but the benefits are lower, as SQLite uses a custom page-management mechanism.

5. RELAX DURABILITY - ACI-RD

When the energy budget is critically low, the durability principles discussed in Section 4 are not sufficient. In other words, always using a strong ACID approach in end-user devices can substantially drain the battery power preventing application from running to comple-

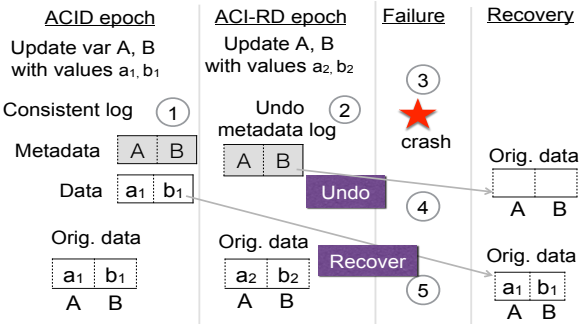


Figure 7: ACI-RD design and steps

tion. This not only impacts ACID-based applications, but also other non-ACID/background applications too.

Key Idea. In a strict ACID-model, the application data, and its metadata, and library metadata, such as persistent allocator state, are logged to a consistent log. For energy critical conditions, we propose a relaxed durability model – ACI-RD, that reduces the frequency/increases the interval between application data logging, but without delaying the associated metadata logging critical for correctness. The application data refers to data generated by application-specific code, whereas metadata refers to any information about the application data such as the NVM address and the bytes of data that are updated, and more importantly the persistent library data which includes memory allocation or removal/deletion information. Section 1, and Figure 1 discussed the difference for a hashtable example. Note that, when the durability is relaxed, the metadata is written to a separate UNDO log. Given the above, an application can transition from a strict ACID phase to ACI-RD and vice versa depending on the available energy. We term these phases as 'epochs' – a fixed interval of time when either ACID or ACID-RD can be used.

Why use a separate ACI-RD metadata UNDO log? The metadata is critical for correctness because in the event of failure during an ACI-RD epoch (the interval when the application data is not made durable), all metadata updates in the UNDO log are used to revert the changes, and restore the last checkpoint state where both data and metadata were durable to a consistent log. Specifically, in a memory persistence-based logging, where the UNDO log contains both application- and library-related metadata, the UNDO log provides information to garbage collect the allocated memory in ACI-RD epoch. Further, it clearly segregates the consistent updates before an ACI-RD epoch from the relaxed updates during ACI-RD. All changes during ACI-RD can be reverted, and just using one log will increase the complexity of sequentially parsing the entire log and classifying consistent and ACI-RD updates.

Transition from ACI-RD to ACID. During transition to ACID, EAP first enables load and store fences, flushes all the data updates from cache-lines, followed by flushing the metadata updates and a load-store fence, so as to guarantee all previous phase updates complete. This provides the same correctness guarantees as an ACID. Also, in-flight errors can exist in both ACID or ACI-RD, and can be avoided if future hardware provides an acknowledgment, as proposed by [12].

ACID-RD Steps. Figure 7 shows the update and recovery steps for two variables A,B in both ACID and ACI-RD epochs. When ① the variables A,B are updated with values a_1,b_1 in an ACID epoch with no energy constraints, both data (a_1,b_1), and the metadata (address of A,B, and size of update) are written to a consistent log. Next, ② shows the case when energy becomes a constraint, for updates of variables A, B with values a_2,b_2: ACID-RD is enabled, and only the metadata (A, B address and update size) is written to ACI-RD UNDO log. Note that, the data is updated in place with FLUSH or atomic updates if applicable. In ③ a failure happens, and during restart and recovery in ④ the updates to A,B from ② are reverted with the UNDO metadata log, and in ⑤ A, B is restored with values a_1,b_1 in the consistent log. The result of this mechanism is a trade-off between durability (D in ACID) and energy, without compromising application correctness. Intuitively, as the ACI-RD epoch time interval increases, the data for which data durability is relaxed also increases, and hence a failure during the ACI-RD epoch increases restart cost. More formally, after a failure in the ACI-RD epoch, the restart time (R_{ACI-RD_t}) is approximately the sum of time to undo all updates in the ACI-RD epoch($UNDOt$), recover data from the consistent log, which is equal to ACID-based restart time (R_{ACID_t}), and recompute ACI-RD transactions which is approximately the transactions lost in an ACI-RD epoch failure ($N_{Trans_{epoch}}$) times the average per-transaction time ($Trans_t$). $UNDOt$ directly depends on $N_{Trans_{epoch}}$, and average UNDO time per transaction ($UNDO_{Trans_t}$).

$$\left.\begin{aligned} R_{ACI-RD_t} &\approx R_{ACID_t} + UNDO_t + (N_{Trans_{epoch}} * Trans_t) \\ UNDO_t &\approx N_{Trans_{epoch}} * UNDO_{Trans_t} \end{aligned}\right\} \quad (2)$$

To implement the ACI-RD mechanism, given an energy budget (E), application execution is divided into per-epoch intervals of t_{msec} with per-epoch energy budget E_{budget}. After the end of each epoch, the increase in epoch energy usage relative to its budget ΔE_{epoch} is estimated, in addition to the increase in the number of transactions performed by the application. As discussed in Section 3, the total energy for persistent applications is a factor of application execution, data logging, cache flush, and metadata transactions. Then:

$$\left.\begin{aligned} \Delta Trans_{epoch} &= Trans_{epoch_i} - Trans_{epoch_{i-1}} \\ \Delta E_{epoch} &= (E_{epoch_i} - E_{budget})/E_{budget} \end{aligned}\right\} \quad (3)$$

where $\Delta Trans_{epoch}$ is the increase in the transactions between epochs i and $i-1$. E_{epoch_i} denotes the energy consumed at epoch i. We assume E_{budget} is known about an application that only commits metadata for correctness (or) by sampling an epoch at runtime [35]. The choice of using this simple estimation is to avoid overheads (including energy) from use of a more complex model.

EAP Execution. Algorithm 1 shows EAP's sequence of steps for reducing energy usage. EAP's efficient and relaxed durability are activated only when energy saving mode is enabled – a feature available in most end-user

```
if energy_save_mode = true then
    Switch to undo logging
    Apply EAP batch allocation
    Apply EAP delayed garbage collection
    for each epoch do
        find ΔTrans_epoch from Equation 3
        if ΔTrans_epoch <= 0 then
            continue;
        end
        if energy_critical = true then
            if ΔTrans_epoch > 0 then
                apply_relaxed_logging();
            end
        else
            if commit.size < cacheline.size then
                atomic commit;
            end
            if commit.size > maxsize then
                Apply ACID-RD();
            end
        end
    end
end
ACID-RD():
Apply fence
Update data in place, FLUSH, drain
Log metadata to UNDO log
```

Algorithm 1: EAP steps: transaction in a new epoch

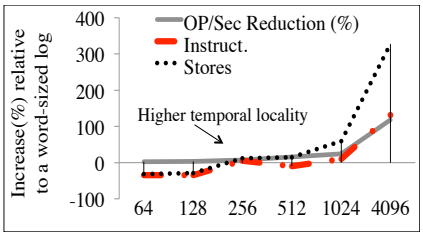


Figure 8: Log size impact, x-axis: per-log size(bytes)

device. When the energy budget is low, but not critical, EAP enables the energy efficient durability principles described in Section 4, which include switching from WAL to UNDO logging, batched allocation and relaxed garbage collection, while still maintaining ACID guarantees. We refer to this mode as *EAP-ACID*. In the next epoch, the EAP runtime checks if EAP-ACID is sufficient to meet the per-epoch energy budged, and if not ACI-RD is activated. These steps are repeated for subsequent epoch until the energy budget is met.

Transparent atomic instructions. Recent Intel architectures support cache-line atomic instruction such as CLWB for cache-line write back (no invalidation; ordered with respect to store fences), PCOMMIT (commits writes to persistent memory or non-temporal stores), and CLFLUSHOPT (optimized cache-line flush) [36]. To exploit this, when, only EAP-ACID is used, the atomic instructions are used for data and metadata updates to log. Direct in-place update is only possible when all updates within a transaction are not larger than cache-line to avoid inconsistent state between actual data location and log [32]. When ACI-RD is used, atomic commits are used for the metadata log as shown in the Algorithm 1.

Relaxing large data updates. When logging small recently modified data, due to temporal locality, data is moved from processor cache to NVM, instead of NVM-to-NVM movement for large updates, and the logging costs significantly increases as shown in Figure 8. Hence, under energy constraints, for transactions with large data updates beyond a threshold (1024B in our evaluation), the durability is relaxed for the entire transaction (ACI-RD works for one or more transactions). While this is not useful for data structures with several small updates (e.g., B-tree), for applications such as snappy with large multimedia file compression in a transaction, logging cost can be significantly reduced. Algorithm 1 shows the logic.

6. EAP EVALUATION

The key goal of our evaluation is to understand the impact of the proposed energy efficient durability (EAP-ACID), relaxed durability (ACI-RD), and the overall benefits of EAP that combine EAP-ACID and ACID-RD. Before discussing the measurements, we briefly describe the methodology of evaluating the CPU and NVM energy, and our assumptions and baseline.

Methodology. Our runtime dynamically estimates the increase in ACID-related energy usage, and based on the available energy budget, adapts and applies the step-by-step optimizations described earlier. Hence, it is important for the profiler and ACI-RD logic to be unobtrusive concerning their own consumption of CPU instructions and NVM access. To enable this, we implemented an in-house driver that provides APIs to selectively calculate the increase in CPU instructions, NVM access, CPU, and NVM energy for a set of transactions in a epoch. This is internally done by accessing the hardware performance counters. Next, modern processors are capable of providing CPU-related energy and power by exporting the information to MSR registers with a minimal sampling rate of less than 10 msec (RAPL [37]). For estimating the NVM energy, we use the NVM load-store information, and PCM load-store cost (40x, and 2x higher write, read energy than DRAM) from [15]. Note that, we are only interested in the relative increase in energy from ACID and other proposed methods, hence the design principles, and evaluation trends, should be applicable for other NVMs also.

Baseline and assumptions. For the evaluation of energy efficient and relaxed durability, we compare the full EAP approach, which combines the energy efficient durability (EAP-ACID) with ACI-RD, against the following durability methods: ① FLUSH: As discussed in Section 3, just flushes, fences, and orders data from cache without any durability, and does not satisfy correctness. We use this as the baseline. ② Metadata-only: This combines FLUSH and metadata logging, but with data durability. An energy and performance optimal case without affecting the correctness, but lack of any data durability guarantees make it less useful. ③ ACID: Strict durability mechanism, that uses WAL for durability, logging data and metadata for each transaction. ④ EAP-ACID: An energy aware ACID approach that uses the efficient durability principles (Section 4), and energy-aware allocation and garbage collection.

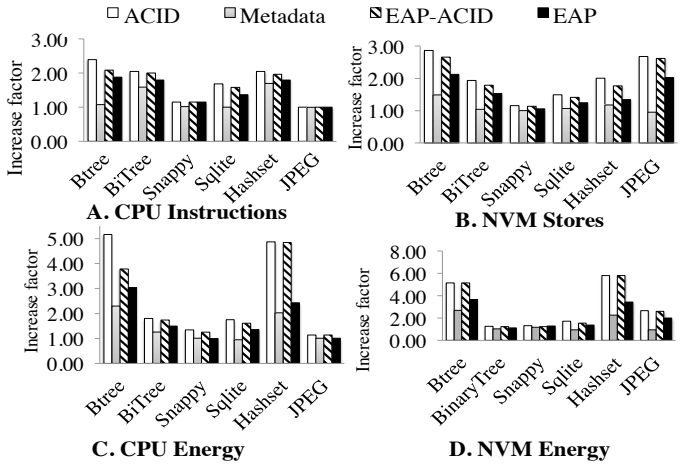


Figure 9: EAP impact:Y-axis shows increase factor(x) relative to FLUSH

6.1 Instruction, NVM access, and Energy

Figures 9.(A)-(D) report the implications of persistence support on increase in CPU instructions, NVM writes, CPU energy, and NVM energy respectively, with the Flush only approach as the baseline. The experiments use the applications and benchmarks introduced in Section 3. For these analysis, the applications runtime vary between ∼28-64 seconds. In EAP-ACID and EAP, the epoch time is set to 450ms, and we later discuss the epoch interval sensitivity. All tree data structures perform insert, find, delete and overwrite operations with 128 byte values. Firstly, we observe that for the FLUSH approach the persistence overhead is relatively small compared to ACID, as expected since this approach does not provide any durability. Similar results was observed by S. Zhao et al. [6], when using a nonvolatile cache. The observations concerning the other methods can be explained in the context of the specific applications as follows.

B-tree: For persistence efficient B-tree, compared to the baseline, the CPU-instruction and NVM access overheads for ACID are 2.40x, and 2.86x, vs. 2.09x and 2.66x for EAP-ACID. EAP-ACID, by trading capacity with energy reduces the ACID overheads by 18%, and switching to UNDO logging, which specifically benefits search (read) operations, provides additional 14% benefits. By combining EAP-ACID with ACI-RD, EAP improves the benefits 69% in terms of CPU instructions, and 66% NVM access, whereas, CPU and NVM energy reduce by 64.7%, and 40.4%. While we study the sensitivity of epoch interval and energy budget shortly, we observe here that even with 400-500ms interval between relaxed and strict ACID-modes, for a cache efficient data structures like B-tree, significant reduction in energy use at critical levels can be achieved.

Binary tree: Binary tree's are highly persistence inefficient with worst case of O(n) compared to B-tree's O(log n). As the tree depth increases, most time is spent on navigating across the tree for inserts, delete and search operations. This explains the less than 36% CPU instructions, and 64% NVM access difference between even the metadata-only and ACID approach, where most difference comes from avoiding cache inefficient log update – each node update or delete results in flushing and logging two other child pointers (16-bytes on a 64-bit machine). EAP-ACID reduces the overall cost only by ∼10%, mainly because, unlike B-tree where node deletions happen even during insert operations due to re-balancing, binary tree only grow unless there are explicit node deletions. EAP reduces the CPU instruction and NVM access by 18% and 16%, respectively.

Hashtable: For persistent hashtable, EAP reduces the high CPU and NVM energy costs of ACID by 2x and 2.5x, respectively. EAP still exhibits high overheads relative to the baseline. This is because the keys, values, and the hash index are all flushed and logged for every update, which can be reduced by further relaxing data durability, or by redesigning the persistent hashtable data structure to reduce ACID cost.

Snappy and JPEG: In case of Snappy [27], with 1.5 GB workload, ∼50K files, and file size varying between 10KB-140MB, the metadata durability cost is low because of low *metadata/data* size ratio. However, given larger data size relative to tree benchmarks, even the FLUSH approach is expensive. EAP, by relaxing logging for large data updates (see Algorithm 1), provides good benefits. From our experience in supporting memory-based persistence for these applications, we realize that using strict ACID-based logging for intermediate compression steps is unnecessary because, after a failure, only files for which compression failed can be re-compressed. While developers can provide application-specific logging customization, EAP transparently provides this support. JPEG image conversion [28] showed similar trends with only 40K transactions, but we observe higher NVM store access, and NVM energy mainly because, unlike SNAPPY, the conversion output can be up to 30% larger for some images, requiring higher logging cost

SQLite: The SQLite benchmark has highly varied access patterns and data granularity as analyzed in Section 3. We observe that by supporting flexible logging and allocator optimization, EAP-ACID instructions and NVM writes are reduced by 16% and 12%, respectively, compared to the default ACID with WAL-based logging. Relaxing the persistence guarantees to ACI-RD leads to around 28% and 26% NVM and CPU energy benefits, respectively. However, the gains are limited for the page-based logging in SQLite due to redundant writes of page [31] even if one object in a page is modified. Finally, new UNDO-log creation during ACI-RD, and the logic to route new transactions add ∼11% overhead. Redesign of SQLite for memory-based persistence, can avoid such cost.

6.2 Energy budget, Epoch interval, and Restart

We next the discuss the sensitivity of EAP gains on the available energy budget, followed by the impact of epoch interval, and the restart cost for the persistent efficient B-tree.

Energy budget: Figure 10 shows the impact of energy budget, with x-axis showing the increase (%) in the budget relative to the metadata-only case, and the y-axis showing the gains relative to strict ACID. Intu-

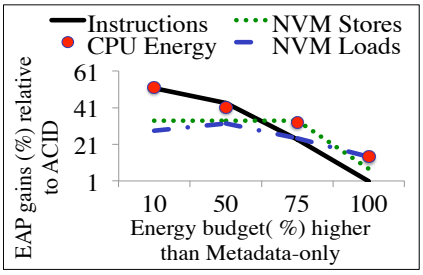


Figure 10: Impact of Energy budget

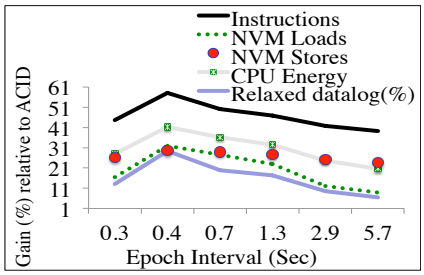


Figure 11: Impact of epoch interval

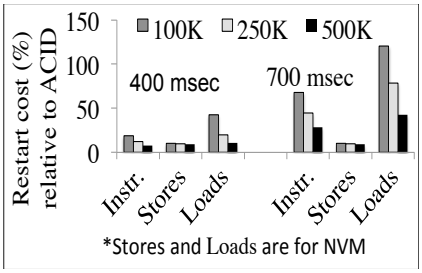


Figure 12: Epoch interval vs. restart Cost

itively, when the budget is high, the number of ACI-RD epochs is lower, and hence the gains too. As observed, for 0-45% increase in the budget, EAP gains – CPU instructions and energy (∼56%-40%), and NVM accesses (∼36-31%) are relatively similar, which shows the (1) significant difference between the ACID and metadata-only case, and also (2) the need to use both efficient durability (EAP-ACID) and ACI-RD for most epochs when the budget is low. When increasing the budget beyond 50%, ACI-RD epochs decreases, and at ∼90%, EAP's mechanisms are turned off to use ACID.

Epoch interval: Figure 11 shows the sensitivity to the epoch interval duration. Note that, after each epoch interval, EAP's algorithm 1 is used to decide if EAP-ACID and/or if full EAP should be applied. Apart from instruction and NVM access gains, we show the relaxed datalog size(%), relative to ACID, for B-tree. We observe the following. First, EAP achieves maximum gains for 400ms-700ms interval range (≈10K-50K B-tree transactions). Further, reducing the interval to 100ms, or increasing to 1-5 seconds, reduces the gains. This is because shorter interval increases errors in not relaxing epoch's that should have been, and importantly, a significant cost is EAP's software-based profiling overhead (up to 10% more instructions). Increasing the epoch interval beyond 1 second fails to apply ACI-RD when the energy budget exceeds the epoch budget, thus incurring higher data durability cost, confirmed by the same trends for relaxed datalog(%) by ACI-RD for the B-tree benchmark with fixed size 128-bit values. *These results confirm the high data logging cost, and show that even by relaxing the data logging to 400ms, reductions of up to ∼59.2% CPU instructions and ∼40.4% NVM access can be achieved with EAP.*

Restart: As discussed in Section 5, and Eq. 2, the restart cost is dependent on the epoch interval for which data durability is relaxed. For brevity, we show the impact of 400ms and 700ms epoch intervals in Figure 12. The y-axis shows the increase in restart cost relative to strict ACID, and x-axis shows the increase in instructions, NVM stores and loads for 100K, 250K, and 500K transactions. Firstly, as expected, for a 400ms epoch interval (∼5K transactions are relaxed) and 100K transactions, the restart instruction, NVM stores, and load is ∼19%, 10%, 43% respectively, whereas for 500K, the overheads are even lower (8%, 9.4%, and 11.67%). The difference between 100K and 500K is because, the cost of loading the UNDO and consistent log, and other initialization dominates, and the cost amortizes with 500K case. With 700ms interval (∼12K relaxed transactions), the overhead of CPU instructions and NVM loads during restart increases. This is mainly because UNDO of persistent node deletions are expensive as they require a new allocation, data copy from the consistent log, and also logging them. The cost increases with increasing node deletions during ACI-RD. However, note that, the CPU energy overhead is less, because the recovery time is around 59x less that actual application runtime.

Summary and Discussion. In summary, EAP, by combining EAP-ACID and ACI-RD provides up to 64%, 40% CPU and NVM energy gains over ACID even for persistence efficient B-tree, and ∼2x benefits for hashtable. Although EAP-ACID provides around 6%-31% gains over ACID, for energy critical conditions, ACI-RD is required for applications run to its completion. EAP does not compromise on metadata durability for correctness. Hence for, applications with large number of small updates and allocations, application redesign, developer hints for batching updates such as group commit protocol [38], H/W support for multi-cacheline commits, dynamic H/W-based epoch energy profiling, and throttling can reduce metadata and software energy cost.

7. CONCLUSIONS AND FUTURE WORK

This paper analyzes the energy overheads of persistence, specifically identifying durability costs as the most significant contributor to energy usage. To reduce durability cost, we propose energy-aware persistence (EAP) that first trades-off a small fraction of performance with flexible logging, and NVM capacity with delayed garbage collection under low energy budgets. For critically-low energy conditions when such optimizations are insufficient, EAP proposes a relaxed durability (ACI-RD)that trades off data durability for energy, without affecting application correctness. EAP is evaluated with representative end-user benchmarks and applications, and shows up to 2x energy reduction supported by reduced CPU instructions, NVM access and energy use relative to a strict ACID approach. An interesting additional outcome of this research is that, to reduce persistence cost, it is not only important to reduce NVM access energy, but to also the CPU energy overheads from the software stack enforcing ACID properties. Interesting future work is to understand the failure models of application and end-user devices, which can be used for supporting alternative durability models without trading off durability or capacity. We plan to analyze of the proposed methods across complex multiple stacks of Android-based devices, and analyze more applications.

8. REFERENCES

[1] "Intel-Micron Memory 3D XPoint." http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html.

[2] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *SIGARCH Comput. Archit. News*, vol. 37, pp. 24–33, June 2009.

[3] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang, "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, (New York, NY, USA), pp. 455–470, ACM, 2014.

[4] H. Kim, M. Ryu, and U. Ramachandran, "What is a good buffer cache replacement scheme for mobile flash storage?," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, pp. 235–246, June 2012.

[5] S. Kannan, A. Gavrilovska, and K. Schwan, "Reducing the cost of persistence for nonvolatile heaps in end user devices," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 512–523, Feb 2014.

[6] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "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, (New York, NY, USA), pp. 421–432, ACM, 2013.

[7] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, (New York, NY, USA), pp. 133–146, ACM, 2009.

[8] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 265–276, IEEE Press, 2014.

[9] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pp. 216–223, Oct 2014.

[10] L. Sun, Y. Lu, and J. Shu, "Dp2: Reducing transaction overhead with differential and dual persistency in persistent memory," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, (New York, NY, USA), pp. 24:1–24:8, ACM, 2015.

[11] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, (New York, NY, USA), pp. 15:1–15:15, ACM, 2014.

[12] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 385–395, IEEE Computer Society, 2010.

[13] "Numonyx PCM projection." http://bit.ly/1sph5Qz.

[14] H. Yoon, "Row buffer locality aware caching policies for hybrid memories," in *Proceedings of the 2012 IEEE 30th International Conference on Computer Design (ICCD 2012)*, ICCD '12, (Washington, DC, USA), pp. 337–344, IEEE Computer Society, 2012.

[15] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 2–13, ACM, 2009.

[16] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 91–104, ACM, 2011.

[17] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 105–118, ACM, 2011.

[18] Intel, "Logging library." https://github.com/pmem/nvml.

[19] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 401–410, ACM, 2012.

[20] J. Zhao, O. Mutlu, and Y. Xie, "Firm: Fair and high-performance memory control for persistent memory systems," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, (Washington, DC, USA), pp. 153–165, IEEE Computer Society, 2014.

[21] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage management in the nvram era," *Proc. VLDB Endow.*, vol. 7, pp. 121–132, Oct. 2013.

[22] D. Sengupta, Q. Wang, H. Volos, L. Cherkasova, J. Li, G. Magalhaes, and K. Schwan, "A framework for emulating non-volatile memory systemswith different performance characteristics," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, (New York, NY, USA), pp. 317–320, ACM, 2015.

[23] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using rapl," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, pp. 13–17, Jan. 2012.

[24] Intel, "Persistent binary tree." https://github.com/pmem/linux-examples.

[25] "SQLite." http://www.sqlite.org.

[26] Google, "LevelDb." http://leveldb.org/.

[27] Google, "Snappy Compession." http://tinyurl.com/ku899co.

[28] "JPEG library." http://libjpeg.sourceforge.net/.

[29] "Binary tree performance." https://groups.google.com/forum/#!topic/pmem/Dqq4FXDFRiY.

[30] "SNIA specification." http://www.snia.org/sites/default/files/Paul_von_behren_NVML_Implementing_Persistent_Memory.pdf.

[31] G. Oh, S. Kim, S. Lee, and B. Moon, "Sqlite optimization with phase change memory for mobile applications," *PVLDB*, vol. 8, no. 12, pp. 1454–1465, 2015.

[32] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, "From aries to mars: Transaction support for next-generation, solid-state drives," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 197–212, ACM, 2013.

[33] K. Paul and T. K. Kundu, "Android on mobile devices: An energy perspective," in *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, (Washington, DC, USA), pp. 2421–2426, IEEE Computer Society, 2010.

[34] M. Hertz and E. D. Berger, "Quantifying the performance of garbage collection vs. explicit memory management," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, (New York, NY, USA), pp. 313–326, ACM, 2005.

[35] I. Constandache, S. Gaonkar, M. Sayler, R. Choudhury, and L. Cox, "Enloc: Energy-efficient localization for mobile phones," in *INFOCOM 2009, IEEE*, pp. 2716–2720, April

2009.

[36] Intel, "Intel Development Manual." http://intel.ly/1CdHj1r.

[37] "Intel RAPL driver." http://lwn.net/Articles/545745/.

[38] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage management in the nvram era," *Proc. VLDB Endow.*, vol. 7, pp. 121–132, Oct. 2013.