KLOCs: Kernel-Level Object Contexts for Heterogeneous Memory Systems

Sudarsun Kannan
Rutgers University
USA

Yujie Ren
Rutgers University
USA

Abhishek Bhattacharjee
Yale Univeristy
USA

ABSTRACT

Heterogeneous memory systems promise better performance, energy-efficiency, and cost trade-offs in emerging systems. But delivering on this promise requires efficient OS mechanisms and policies for data tiering and migration. Unfortunately, modern OSes are lacking inefficient support for data tiering. While this problem is known for application data, the question of how best to manage kernel objects for filesystems and networking—i.e., inodes, dentry caches, journal blocks, socket buffers, etc.—has largely been ignored and presents a performance challenge for I/O-intensive workloads. We quantify the scale of this challenge and introduce a new OS abstraction, *kernel-level object contexts (KLOCs)*, to enable efficient tiering of kernel objects. We use KLOCs to identify and group kernel objects with similar hotness, reuse, and liveness, and demonstrate their use in data placement and migration across several heterogeneous memory system configurations, including Intel's Optane systems. Performance evaluations using RocksDB, Redis, Cassandra, and Spark show that KLOCs enable up to 2.7× higher system throughput versus prior art.

CCS CONCEPTS

• **Software and its engineering → Virtual memory**.

KEYWORDS

Heterogeneous Memory, OS, Nonvolatile Memory, Virtual Memory

ACM Reference Format:
Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. 2021. *KLOCs*: Kernel-Level Object Contexts for Heterogeneous Memory Systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3445814.3446745

1 INTRODUCTION

Memory heterogeneity is here. Emerging systems combine the best properties of memory technologies optimized for latency, bandwidth, capacity, persistence, and cost. Multiple DRAM nodes are being augmented with die-stacked DRAM [15, 30, 45], high-bandwidth multi-channel DRAM (e.g., Intel's Knight's Landing [6]), and byte-addressable NVMs (e.g., 3D XPoint in Intel Optane DC) [4, 14, 16].

While heterogeneous memory systems may offer better performance, energy-efficiency, and cost trade-offs, they complicate memory management. Decades of research have demonstrated the challenge of data allocation and migration in multi-socket non-uniform memory access (NUMA) architectures [7, 8, 10, 26, 33, 47]. Heterogeneous memory systems amplify this challenge by integrating memory devices with more varied latency, bandwidth, and capacity characteristics.

To optimize a heterogeneous memory system for performance, one would ideally place the hottest data in the fastest memory node (in terms of latency or bandwidth) until that node is full, the next-hottest data would be filled into the second-fastest node up to its capacity, and so on. As a program executes, its data would be periodically assessed for hotness and re-organized to maximize performance. For emerging software-controlled heterogeneous memory systems, hotness detection and migration requires effective software mechanisms and policies to determine data reuse and control data migration. While it is possible for application developers to orchestrate these tasks, efficient OS approaches that are transparent to the programmer are preferable because of their less onerous programming model. Current OS mechanisms to measure reuse and migrate data have, however, surprisingly high overheads and have consequently been the subject of recent software and hardware acceleration techniques [13, 19, 31, 33, 35, 37, 40, 50, 53, 57].

Unfortunately, most prior research on OS-directed data tiering focuses on application-level data and ignores kernel objects. One exception is recent work that migrates and replicates page tables in DRAM devices in different sockets [11], but memory tiering of kernel objects for storage and networking I/O remains unexplored. This is because kernel objects have traditionally been thought to be few in number, restricted in memory footprint, and less significant in their impact on overall performance. This view is driven by network and disk I/O speeds that are several orders of magnitude slower – and hence more consequential to performance – than memory. But while this was true in the past, advances in networking and storage speeds now make memory management of kernel objects critical to performance. We quantify the scale of this criticality by showing that current approaches that ignore tiering of inodes, dentry caches, journal blocks, network socket buffers, etc., leave as much as 4× performance on the table. This paper's central contribution is to recover this wasted performance via a new OS abstraction, *kernel-level object contexts (KLOCs)*, that permits fluid tiering of kernel objects.

The KLOC abstraction: KLOCs are logical groupings that capture the kernel objects associated with OS entities requested by applications. Kernel entities requested by applications are files and sockets,

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446745

Table 1: Kernel objects associated with file systems and networking that form the basis of this work.

KLOC Type	Kernel Object Structure
FS/Network	*inode_struct* - Per-file inode
FS	*block* - Block I/O structure for conversion of metdata to disk blocks
FS	*journal* - Filesystem journal buffers
FS	*page_cache* - Buffer cache page
FS	*dentry* - Name resolution for each file
FS	*extent* - Structure for grouping contiguous disk blocks
FS	*blk_mq* - Block layer multi-queue structure for parallel dispatch of blocks to disk
Network	*sock* - Socket object for packet buffers
Network	*skbuff* - Header for packet buffer
Network	*skbuff->data* - Data buffer for packet
Network	*rx_buf* - Network receive driver buffer

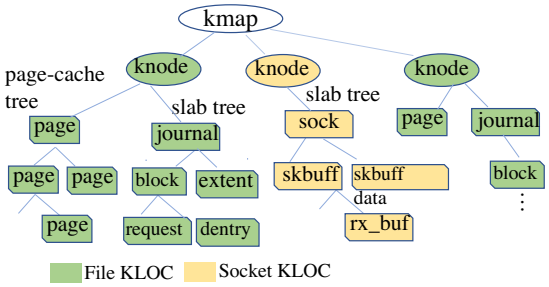


Figure 1: All of the kernel objects associated with each active file and active socket represent individual KLOCs. All the KLOCs in the system are tracked using a kmap. The inode of each active file or socket maintains a pointer to a knode data structure, which tracks associated kernel objects.

while kernel objects range from structures associated with files (e.g., inodes, blocks, journals, etc.) to those associated with sockets (e.g., packet buffers, headers, data buffers, etc.), as listed in Table 1. Figure 1 shows that we treat all the kernel objects associated with each active file and each active socket as individual KLOCs. This means that in Unix-based "everything is a file" OSes, there is one KLOC of kernel objects associated with each inode.

Using KLOCs: Good performance is achieved when hot application data and kernel objects are placed in faster and nearer memory up to its capacity, while colder data and kernel objects are placed in slower and more distant memory. As we will show, kernel objects are rapidly allocated and deallocated, and have much shorter lifetimes than application pages. This makes it challenging to extend existing OS LRU code paths that identify hot/cold pages – originally built for longer-living application pages – to place and migrate kernel objects in a sufficiently timely manner for good performance. Even if some of these code paths could be accelerated, the diverse assortment of kernel objects used today and the complexity of their intertwined memory allocation, reuse, and deletion code paths make it difficult to implement these changes.

KLOCs offer a principled way to tame this diverse ecosystem of kernel objects and quickly ascertain their hotness/coldness. When the OS determines that an inode has become cold (because, for example, the file or socket associated with the inode has been closed), KLOCs permit direct identification of all kernel objects associated with the inode and mark them as candidates for migration to slow memory. In other words, rather than relying on expensive and independent traversals of separate code paths for all the kernel objects to gradually which of them are cold, KLOCs short-circuit this process and migrate related cold kernel objects en masse. Our implementation in a Linux 4.17 kernel shows that KLOCs improve the performance of I/O-intensive workloads like RocksDB, Redis, Cassandra, and Spark by up to 2.7× on a two-tier memory system and 1.4× on a multi-socket Intel Optane system compared to state-of-the-art application tiering (*Nimble* [53]).

Implementing KLOCs: In realizing KLOCs, we answer several important research questions:

What OS entity should KLOCs be anchored to? Grouping kernel objects according to files and sockets strikes a good compromise between performance and minimal kernel changes. This is because it allows identification of well-defined points where the OS can manipulate kernel objects – i.e., existing system calls for file and network I/O (e.g., file create, open, etc.) – and also naturally groups related kernel objects. Leaning on existing system calls also means that KLOCs are transparent to programmers and manipulated entirely within the kernel.

How should KLOCs manage member kernel objects? The OS must group millions of kernel object pages with diverse sizes, reuse, and intricate associations across files and sockets into KLOCs. We rely on principled use of data structures already widely employed in real-world OS kernels to efficiently track these relationships. Figure 1 illustrates our implementation. Each inode is expanded to maintain a pointer to a knode structure, which uses kernel red-back trees to track all associated kernel objects. When the OS opts to migrate a KLOC, the kernel objects pointed to by the subtree under the corresponding *knode* are migrated. Furthermore, all system KLOCs are tracked using a global *kmap* structure, which maintains pointers to all system knodes.

What changes to kernel object code paths are necessary to support KLOCs? To support KLOCs, some kernel object code paths need to be changed. For example, KLOCs must enable the relocation of kernel objects. Unfortunately, OSes create kernel buffers with either slab allocators, which are fast but preclude kernel object relocation. In contrast, *vmalloc* and *page alloc* allocations permit kernel object relocation but are unsuitable for kernel objects that are referenced by physical address. We create a KLOC allocation interface that permits fast allocation of kernel objects while supporting relocatability and, via systematic study, are able to redirect 400+ allocation sites to our interface. Similarly, associating a kernel object with the right file/socket can be a high-latency endeavor. For example, the OS determines the socket for incoming network packet buffers only after traversing several levels in the TCP stack. This long-latency process can overly delay kernel object migration decisions. We design KLOCs to circumvent these challenges and enable the fast association of kernel objects with files/sockets.

Overall, we show that memory management of kernel objects has become vital to the performance of heterogeneous memory systems. KLOCs are an initial approach to tame the large design space of kernel object management options. We expect future research to improve the efficiency and design elegance of kernel object tiering, but believe that the notion of kernel object *contexts* can help manage the continued growth in memory footprint and diversity of kernel objects.

2 RECENT WORK ON DATA TIERING

Heterogeneous memory devices are being integrated into systems with conventional DRAM, with die-stacked 3D-DRAM, Hybrid Memory Cube (HMC), High Bandwidth Memory (HBM), and byte-addressable NVMs showing early promise in addressing the big-data needs of modern applications [12, 35, 45, 48]. While they offer performance benefits, these devices pose complex performance and capacity tradeoff questions. Technologies like 3D-DRAM, HMC, and HBM provide 2-10× higher bandwidth and 1.5× lower latency than conventional DRAM, but suffer 8-16× lower capacity [12, 15, 16, 41]. Meanwhile, byte-addressable NVMs offer 4-8× higher capacity than DRAM but suffer 2-3× higher read latency, 5× higher write latency, and 3-5× reductions in access bandwidth [26, 44, 52]. To manage a mix of heterogeneous memories, recent studies propose several software and hardware techniques, including OS and runtime approaches [13, 27, 33, 35, 37, 41, 53]. Most of these approaches track page hotness by scanning page tables to migrate hot application pages of different sizes to fast memory. Approaches such as HeteroVisor [27] and HeteroOS [33] propose data placement and migration for applications in virtualized datacenters, while other work accelerates page migration using multi-threading and more efficient caching [53]. Lagar-Cavilla et al. [35] propose a combination of OS-level hotness scanning combined with machine learning for data placement.

In contrast, hardware approaches for data tiering include augmenting the memory controller [19, 46] or the TLBs [40] for efficient identification of hot pages and migration. These studies have primarily focussed on byte-addressable NVMs and on-chip die-stacked 3D-DRAM technologies [15, 18, 19, 25, 28, 31, 36, 38, 40, 42, 57]. While the NVMs are used as slower memory [36], stacked 3D-DRAMs are used either as a hardware-managed last level L4 cache [18, 31, 38, 43, 57] or faster DRAM. The hardware memory controller is delegated with the responsibility of managing page placement across memories as well as predicting and prefetching pages.

None of these studies consider kernel object tiering. Recent work on accelerating OS page migration mechanisms place kernel objects either entirely in slow memory for two-tier memory systems or in DRAM local to the CPU that allocated the kernel object for conventional NUMA systems [33, 53, 54]. They do not quantitatively ascertain the performance impact of these decisions or consider alternatives. The closest prior work comes to studying kernel objects placement is *Mitosis*, a recent study on page tables placement across NUMA memory sockets [11], but even this ignores file or networking objects. In fact, not only is there no prior work on heuristics and mechanisms for file and network object tiering, modern OSes cannot migrate many kernel objects for reasons that we discuss subsequently. While better hardware caching and prefetching techniques complement KLOCs by improving data placement to a faster memory, these techniques do not differentiate between kernel objects and application pages with different lifetimes. We show the need to treat short-lived kernel pages differently from application pages and increase the direct placement of kernel objects by avoiding delays from hotness detection and migration overheads.

3 MOTIVATION

3.1 Prevalence of Kernel Objects

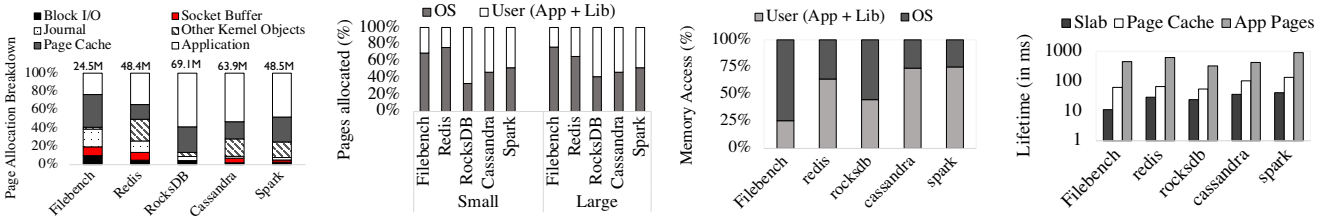
In this section, we characterize the memory footprint, reuse, and lifetimes of kernel objects. In section 6, we summarize the I/O-intensive workloads and platforms used in our studies. Figure 2a shows the percentage of pages allocated to different kernel objects and separates these from application-level page allocations. All workloads are configured with 40GB input data sets, and we quantify the number of pages allocated in units of millions of pages on top of each bar. Kernel objects are prevalent for all these applications. Consider Filebench, which uses 16 threads to read and write 4KB blocks to separate files. Writes and reads to disk may prompt page cache page allocations, updates, and allocation of journals, metadata radix trees, block driver buffers, etc. As another example, consider RocksDB, which updates hundreds of 4MB files with key-value data, and spends 40% of its runtime within the OS kernel allocating inodes, driver block I/O and journals, dentry caches, and radix tree nodes. Spark [56], which uses the Hadoop file system to store and checkpoint data, is similarly filesystem-intensive.[1] These observations apply to network-intensive workloads too. For example, Redis allocates a significant number of kernel object pages for ingress and egress socket buffers, and page cache pages to periodically checkpoint key-value store state to a large file on disk [9].

Figure 2a shows that kernel object memory footprints can rival memory capacities expected for high-bandwidth DRAM devices in the near-term. Recent studies focus on fast memory nodes in the range of 4-16GB [13, 33, 41, 53, 54], and our results show that even with a modest input data set of 40GB, I/O-intensive workloads need more than 10s of GBs for kernel objects alone. Different workloads rely on different sets of kernel objects extensively. For example, while page cache pages dominate RocksDB allocation, Redis and Cassandra require a mix of page cache and socket buffer objects. Overall, kernel objects are plentiful, even exceeding application pages in some cases, and need to be carefully managed.

Figure 2b shows that the observations from Figure 2a hold as the sizes of our input data sets are changed. In Figure 2b, our workloads are adjusted to use 10GB input data sets (Small) in addition to the 40GB input data sets showed in Figure 2a (Large), and we show the percentage of pages allocated to kernel objects versus userspace. Kernel objects continue to use a significant fraction of the total pages.

Figure 2c quantifies the percentage of memory references to user-space data versus kernel objects. These results were collected using on-chip performance counters via Intel's VTune, and Linux Perf [5], and shows that kernel objects are accessed often. Consider a file write in Filebench. The virtual file system looks up the page cache radix tree, allocates a new page if necessary, inserts the page into the radix tree, performs metadata/data journalling, and finally, commits blocks to storage. These steps are even more memory-intensive than writing data to the page cache because of the increase in random accesses and poor locality of reference. In fact, scaling the workload inputs leads to a sharp increase in LLC misses due to higher traffic to kernel buffers. Filebench spends 86% of execution

[1] The Hadoop filesystem is run as a separate process that maintains user-level caches and periodically updates page caches.



(a) Percentage of total memory footprint for different kernel objects and application pages. Raw page counts shown at the top of each bar. Workloads scaled to 40GB input data set.

(b) Percentage of the memory footprint for kernel objects (OS) versus application data (App + Lib). Small/large x-axis labels correspond to workloads with 10GB/40GB input data set.

(c) Percentage of memory references dedicated to kernel objects (OS) versus applications (user-space App + Lib). All workloads are scaled to 40GB input data set.

(d) Lifetime of important kernel objects versus application pages. The y-axis is in log scale. Slab and page cache pages are kernel objects, and are shorter-lived than application pages.

Figure 2: Figure 2a shows the breakdown of pages used by application, page cache, and other slab allocations across the file and the network subsystems for the large workload. In Figure 2b, the y-axis shows the percentage of page allocations in the application and the OS. Small and large workloads use data sizes (RocksDB, Redis, and Cassandra) or file sizes (Filebench, Spark) of 10GB and 40GB, respectively. OS allocations include page cache, slab, and *vmalloc* objects. Pages are allocated and released (freed) frequently; hence the total allocations can be greater than available memory. Figure 2d shows the lifetime of application pages, slab, and page cache pages.

time inside the OS, and hence, the memory accesses increase are higher than RocksDB (54%) and Redis (38%).

3.2 Kernel Object Hotness

In section 3.1, we showed that kernel objects are allocated and accessed frequently. Consequently, we need to identify the ones that are hot for placement in capacity-constrained fast memory. We define hot kernel objects as those currently in use by applications or have been recently used, and cold kernel objects as those that are good candidates for placement in slow memory. There are several reasons that a kernel object may become cold. Consider the case when an application closes an open file. If no other processes continue to leave the file open, then the inode, block, dentry, extent, page cache pages, and other structures associated with the file are now cold. As another example, even if the file remains open, it may have been accessed long ago and is hence cold.

As with application pages, there is no clear threshold as to how long ago a file must have been accessed to be considered cold. Rather, to tier kernel objects appropriately, the OS-level LRU policies must be augmented to identify kernel objects associated with files that are definitely cold (i.e., because the file has been closed) and must be able to infer the relative ages of files that have not yet been closed to identify those that are likely cold because of a lack of recent use. The exact number of kernel objects that are to be migrated and the threshold where the kernel objects are considered cold hence remains a function of the OS LRU policy. This notion of kernel object coldness has three implications:

First, on file creation, the associated kernel objects should be allocated in fast memory because they are hot. As they become colder, they may be migrated to slower memory. This presents a contrast to all recent work [41, 53], which, for two-tier systems, allocates kernel objects entirely in slow memory, or, in traditional NUMA systems, allocates them to the memory socket local to the CPU performing the allocation without the option of migrating them in the future.

Second, kernel objects associated with files that have been deleted or completely unlinked (i.e., their reference count is zero) are not cold, but are instead *deallocated*. They should not be migrated to slow memory and can be deleted.

Third, all kernel objects associated with a file inode are treated as having the *same* level of hotness/coldness and are migrated together. This reduces kernel bookkeeping cost and is appropriate because all kernel objects associated with the inode do tend to be accessed during I/O. However, it is possible that in select cases, some kernel objects may see different reuse attributes. In practice, we find that this happens so rarely that opting for an inode-driven view of all kernel objects offers a simplistic implementation and good performance.

3.3 Challenges of Kernel Object Tiering

One may initially consider extending existing OS LRU code paths to also account for all kernel object pages, currently lacking in modern OSes. OSes like Linux or FreeBSD scan to identify hot and cold pages by traversing an application's page table and visiting all physical frames to mark them as eviction candidates. While one could potentially identify kernel objects in this manner, this approach is successful only if the time taken to identify cold kernel objects is significantly faster than the kernel objects' lifetime. Figure 2d quantifies the lifetime of several categories of kernel objects. Because the lifetime of kernel objects is tied to OS mechanisms used to allocate them, we separate kernel objects into those allocated by slab allocators versus kernel objects like page cache pages, which are allocated via other techniques.

Short-lived kernel objects – i.e., inodes, blocks, dentrys, extents, dir buffers, skbuffs – are typically allocated using slab allocators (*kmalloc* and its variants like *kmem_cache_alloc* in Linux and FreeBSD). Kernel objects allocated with *kmalloc* use only contiguous physical pages for allocation, do not require manipulation of page tables during allocation and release, and cannot be relocated. However, they are allocated quickly. In contrast, separate allocators

are used for large kernel objects like filesystem page caches. Unlike slab allocations, these are mapped into the virtual address space of processes in order to satisfy reads/writes from the application.

Regardless of allocation strategy, Figure 2d shows that kernel objects are short-lived. While application-level data for RocksDB and Redis have lifetimes in the tens of minutes, their slab pages are alive for only 36ms on average. Page cache pages live for marginally longer, averaging 160ms. Such short lifetimes for kernel objects are expected. The lifetime of page cache pages can also vary depending on memory pressure. When available free memory is low, the cache pages are aggressively released to accommodate application allocations. Slab-allocated kernel objects consist of buffers added to radix tree nodes to track file metadata or structures like dentry caches and in-memory journals. These structures are frequently queried, allocated, and deleted when trees are rebalanced, or page cache pages are evicted [32]. Consequently, application pages are long-lived enough to tolerate LRU scan times. For example, we measure the time taken to scan one million pages on our Intel Xeon platform as 2 seconds, corroborating results from recent work [13] – but kernel objects are not, even if the LRU scans occur in the background.

Yet another approach may be one where the concept of NUMA nodes is extended to kernel objects, and associations are built between CPU nodes and the kernel objects that they allocate. This would enable kernel objects tiering close to the CPU that likely uses them. Indeed, this is what modern OSes do – they allocate kernel objects on the NUMA socket corresponding to the core that is responsible for the OS activity leading to kernel object creation. However, while NUMA systems do migrate pages between sockets post-allocation if the traffic from remote sockets increases, kernel objects are never migrated. This leads to performance loss when kernel objects are used asynchronously (e.g., receive path kernel objects associated with sockets or pages invoked via I/O prefetching).

4 DESIGN OF THE KLOC ABSTRACTION

We first provide a brief overview of KLOC followed by the design approach.

4.1 Overview

Figures 1 and 3(a) illustrate the data structures involved in realizing KLOCs within the Linux kernel. At their core, these data structures are manipulated by two general OS sub-components. The first is the OS system call interface, which allocates kernel objects and adds pointers to them in the knodes. The knodes act as a "table of contents" to the locations of all associated kernel objects and sidestep the challenges detailed in section 3.3. Intercepting system calls as the medium for knodes to point to kernel objects ensures that the KLOC abstraction remains transparent to applications.

Figure 3(a) shows that the second OS sub-component necessary for the management of KLOCs involves the data structures used by Linux's LRU code paths to identify hot/cold kernel objects. Most OSes, including Linux, use a data structure to track important per-CPU information for scheduling and resource usage. Our approach is to add, to this data structure, a list of pointers to knodes touched by each CPU. As we discuss in section 4, this data structure acts as a software cache of the bigger kmap structure in Figure 1, and is similar in spirit to several other "fast path" software caches that OSes maintain for page tables, virtual memory area (VMA) trees, etc. The knodes are further associated with a variable for tracking their hotness (*age*) and whether they are active (*inuse*). The code paths that implement OS LRU policies use these per-CPU lists of knode pointers to quickly identify cold knodes. Moreover, without walking the page table to identify all the kernel objects associated with this knode, the kernel can identify objects pointed to by each knode. This permits the LRU engine to short-circuit lengthy page table scans.

The exact number of pages, kernel objects, and KLOCs to migrate depends upon memory pressure and LRU policies to govern the aggressiveness with which data migration must be pursued. The KLOC abstraction does not enforce any constraints on these runtime decisions but instead offers fast migration capabilities for any existing OS policy.

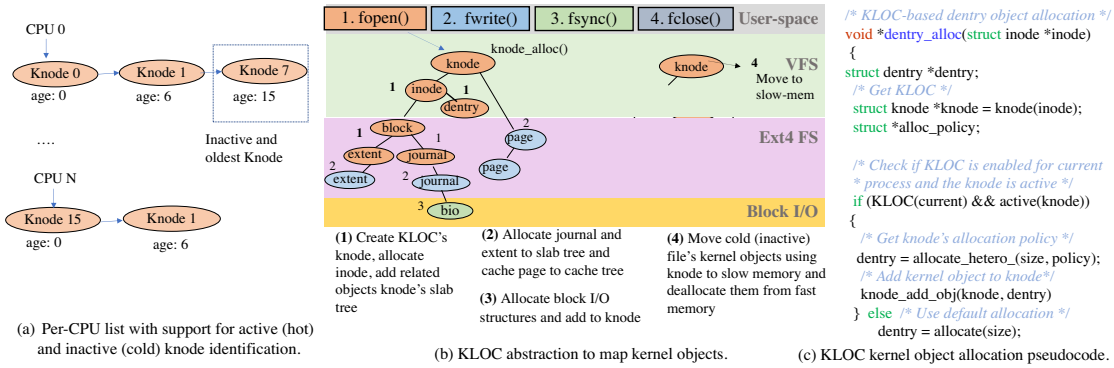
4.2 KLOC Management

Figures 1 and 3 highlight key aspects of the KLOC abstraction. We now discuss more concrete design details. To drive our design, we focus on our prototype in the Linux kernel. Our prototype is likely to extend to other monolithic kernels, even if the implementation details vary. We next discuss how to initiate KLOC, allocation of KLOC's knode, and their management.

4.2.1 KLOC Initiation. System administrators trigger the use of the KLOC abstraction via a *begin_kloc()* system call with the target application passed as an argument. To avoid application changes, we implement this as a shared user-level library that the application can be linked. All activities pertaining to KLOC creation, management, and deletion are handled entirely within the OS and are transparent to the userspace.

4.2.2 Allocation. Every file, whether it is created by the filesystem or the networking stack, has a knode associated with it. Every file's inode maintains a pointer to its associated knode. We make knodes easily accessible to the filesystem and networking system code paths by allocating them within the virtual filesystem (VFS) layer as a red-black tree. We leverage Linux's existing support for red-black trees to enable efficient design, minimize correctness concerns, and ease design effort [23, 34]. When an inode is created, an entry is allocated in the knode red-black tree and pointed to by the appropriate file or socket inode. When an inode is closed, the knode red-black tree is searched, and the appropriate knode is marked inactive. All updates to the knode red-black tree are performed serially to avoid race conditions and deadlocks when multiple CPUs access per-inode RB-trees. In **??**, we discuss how we reduce red-black tree contention and increase concurrency.

We use the slab allocator for knodes in order to optimize for speed of allocation. This is important because I/O-intensive workloads spawn and delete files hundreds of thousands of times over an application lifetime, leading to many knode allocations and deletions. The downside of using slab allocators is, however, that the knodes become non-migratable. However, our profiling results show that prioritizing knode allocation speed over amenability for migration is more important to overall system performance. This is



(a) Per-CPU list with support for active (hot) and inactive (cold) knode identification.

(b) KLOC abstraction to map kernel objects.

(c) KLOC kernel object allocation pseudocode.

Figure 3: On the left (a), we show that existing per-CPU structures in Linux are augmented with a list of knodes accessed by each CPU, with information about the knode reuse. These data structures are used by Linux's LRU engine to identify hot/cold KLOCs. In the middle (b), we show an example of the key kernel objects related to file operations that are managed during *open()*, *write()*, *sync()*, and *close()* operations. On the right (c), we show pseudocode for dentry object allocation using the KLOC abstraction.

because knodes are orders of magnitude fewer in number than the kernel objects that they point to (which must be migratable and cannot, therefore, use slab allocation). Therefore, our design always allocates knodes to fast memory. Note that this is not a fundamental design decision, and other designs are also possible.

Overall, the tight association between inodes and knodes binds KLOC lifetime to that of the file or socket that it is associated with. In other words, when an inode is created, so too is a KLOC. When an inode is deleted, so too is its KLOC.

Next, regarding application pages, KLOCs prioritize application pages to reduce their placement in slower memory, which can significantly impact performance. KLOCs attempt to allocate application pages to a faster memory, unlike kernel objects, where only objects of active knodes are allocated to fast memory.

4.2.3 Associating Kernel Objects to Knodes. After the knodes are allocated, they must maintain pointers to all associated kernel objects. A key research question is the choice of data structure used to track kernel objects. Kernel objects can number in millions (e.g., RocksDB has roughly two million kernel objects). They must be quickly looked up via the knode and tracked using data structures that are correctly implemented. To balance these factors, we opt for Linux's red-black trees. We find that using a single red-black tree to record millions of kernel objects can be prohibitively expensive; empirically, as many as ten memory references are needed on average for tree traversal, posing too high a performance tax. While many design solutions are possible, we use the simple approach of incorporating two red-black trees within each knode – *rbtree-cache* tracks large kernel objects allocated using non-slab allocators, while *rbtree-slab* tracks smaller kernel objects allocated using slab allocators. Beyond its performance benefits, this approach also offers the organizational benefits of separating page cache pages versus smaller kernel allocations.

Any OS subsystem that accesses and manipulates files or sockets is responsible for manipulating the red-black trees. Two such subsystems are system calls for the filesystem and networking system. For example, when a file is created, so are the inodes, dentrys, and journal blocks. A file write can create cache page objects, radix tree nodes, journal records, and extents. Similarly, system calls responsible for socket creation (*socket()*, *open()*) result in creation and manipulation of packet buffers (*skbuff*). When applications invoke egress and ingress activity via *send()* and *recv()* system calls, or when they poll, associated kernel objects are created. In all cases, the pointers to these kernel objects must be inserted or deleted in the target knode's red-black tree. Figure 3(b)-(c) show the diagram and pseudo-code associated with file creation. As shown, an inode is created, a new knode is created, and a pointer to a dentry object is added to the knodes. When the file is written, a page cache page is allocated, and a pointer to it is added to the *rbtree-cache*, while references to the extents and journal records (*journal*) are added to *rbtree-slab*. After the file is closed, the page cache pages are removed. When the file/inode is deleted, so too is its knode. Non-system call OS activity can also change knodes. For example, when the filesystem block driver commits in-memory pages to disk, it allocates the file's block I/O structures. Pointers to these must be added to the appropriate knode.

While identifying the file/socket that a kernel object is associated with is straightforward in many cases, it can pose a challenge in others. Consider the networking stack. Packets are buffered across several layers of ingress and egress paths, including TCP, UDP, IP, and the network device driver (i.e., NAPI). Problematically, the ingress path receives packets asynchronously. As network packets arrive, the device driver allocates a generic packet buffer but does not know the socket to which this packet belongs. This information is extracted in a higher layer of the TCP stack and presents a problem for KLOCs, which need fast association between kernel objects and their corresponding file/socket for maximal performance.

In response, one might extract the packet's entire header to identify the socket inside the driver code before transferring control to the higher TCP layers. We find this to be CPU-intensive and comparable in latency to socket lifetimes, making it infeasible. Instead, we extract socket information within the device driver and eliminate redundant work at the higher-level layers. We do this by extending

the packet buffer structure (*skbuff*) with an 8-byte socket field containing the socket information extracted in the device driver. This field elides the need for further socket information extraction at higher levels of the TCP stack. We also extend the device driver to add packets to the desired knode.

4.3 Concurrency Via Per-CPU Fast Paths

As initially described, we expect our system to suffer from two key sources of degradation pertaining to synchronization. First, multiple threads may simultaneously access per-inode red-black trees, especially when objects are added to knodes or when threads responsible for migrating kernel objects access them. Second, as shown in Figure 1, we use a global kmap implemented as a red-black tree to maintain pointers to all knodes. This global structure is susceptible to synchronization overheads [39].

We exploit Linux's red-black tree with read-copy-update (RCU) support to partly mitigate some of these contention overheads. RCU enables "multi-reader, single-writer" concurrency [20]. To reduce locking overheads, we split a knode's red-black tree into a cache (*rbtree-cache*) and a kernel slab object (*rbtree-slab*) tree. We also, as shown in Figure 3(a), employ a well-known OS approach of creating a "fast path" cache of the kmap by implementing per-CPU linked-lists of associated knodes. Creating separate lists of knodes reduces synchronization overheads, and by restricting their sizes, ensures that they can be traversed fast. However, they pose coherence challenges as the same knode may be accessed by multiple CPUs and may hence be mapped to multiple per-CPU lists. Fortunately, Linux already maintains APIs and mechanisms for coherence management of per-CPU lists [24]; by leveraging these, we achieve correct implementation. We find that existing coherence mechanisms present minimal overhead to the KLOC design. Finally, each per-CPU list associates an *age* variable with the knode pointers as shown in Figure 3a. This age variable is set to zero whenever a knode is accessed and is incremented when the LRU policy scans it but does not mark it as a candidate for eviction. As the age increases, its KLOC becomes colder and becomes a stronger candidate for eviction to slow memory. Finally, KLOCs, similar to other kernel data structures in NUMA-based systems, do not introduce additional false sharing problems via coherence protocols. The combination of per-CPU fast path lists and the red-black trees reduce knode contention. Per-CPU lists reduce the *rbtree-cache* and *rbtree-slab* accesses by 54%. Reusing existing RCU support for red-black trees also minimizes contention among remaining accesses.

4.4 Support for Migration in KLOC

We first discuss the need for supporting kernel object migration and then discuss our support for kernel object and application page migration.

Migrating kernel objects with short lifetime. Cache and slab objects have short lifetimes, but many of them (e.g., inodes, socket buffer structures) are frequently accessed through application lifetime. Using slow memory for all these objects hurts performance (see Figure 5C). KLOCs aim to increase direct allocations of kernel objects of an active knode to faster memory and too significantly "reduce" migration from slow to fast memory. However, migration cannot be completely eliminated because of limited fast memory capacity. In fact, we find that inactive kernel and application pages need to be downgraded from fast to slow memory frequently, and represent 88% of total migrations. Within this group, 79% of the migrations are for page cache pages. KLOCs also permit downgrading of slab objects, which are not freed even after a knode becomes inactive. Because many real-world workloads see inodes, dentry structures, and other filesystem structures enjoying periods of activity interspersed with inactivity, KLOCs are vital to downgrading these structures when necessary. On the other hand, reverse migration from slow to fast memory represents 4-12% of the migrations and is mainly used for cache pages. With increasing fast memory capacity, the slow memory page use reduces, consequently reducing the performance difference across approaches and the variance across workloads.

Finally, we track KLOCs at the inode granularity, as opposed to tracking each object in a fine-grained manner. This enables direct allocation of short-lived kernel objects relevant to an I/O request to fast memory and migration of inactive objects to slow memory. Direct allocation of short-lived kernel objects reduce the cost of moving kernel objects across memories. Our future work will explore the benefits of employing a fine-grained kernel object tracking approach in ways that do not introduce tracking overheads.

Support for Kernel Object Migration. Once the OS kernel identifies KLOCs with cold knodes, it migrates all kernel objects mapped to knodes together. This means that the kernel objects pointed to by a knode subtree in Figure 1 are migrated. While kernel objects allocated using *vmalloc()* and *page_alloc()* (e.g., page cache pages) are relocatable, those that are slab allocated are not. This is because they are not mapped into a virtual address and allow kernel object access using a physical address when required.

While it is possible to make wholesale changes to the slab allocator to fix this, it is a complex endeavor. Instead, we build a new allocation interface for kernel objects, enabling the allocation of kernel objects into virtual address spaces by leveraging existing code paths for anonymous virtual memory area (VMA) regions that are not backed by files. While these VMA regions have traditionally not been relocatable, we found that it was possible to more easily enhance them than slab allocators to support kernel object migration.

Migration of Application Pages. In tandem with kernel objects, application pages deemed to be inactive by Linux's LRU mechanism are migrated to slower memory. In our work, we repurpose OS-level LRU for application data pages, like recent work like Nimble [53], HeteroOS [33], and ThermoStat [13].

Making KLOCs amenable to I/O prefetching: Linux's *adaptive readahead mechanism* prefetches I/O pages with temporal and spatial locality locality [51]. We augment this mechanism to prefetch kernel objects associated with the inode by exposing them to the I/O prefetcher kernel objects via the KLOC abstraction. The I/O prefetcher's existing logic modulates the cost-benefit trade-off of prefetching kernel objects. As we describe in section 7, KLOCs make I/O prefetching even more effective. When the right kernel objects are prefetched, KLOCs enable the I/O prefetcher to identify them more quickly. When the kernel objects are actually poor

prefetching candidates, KLOCs enable the OS to determine that they are cold, to migrate them slow memory more quickly.

4.5 KLOC in HW-SW Managed Tiering

The KLOC abstraction is usable by any existing kernel-level policy that tiers data. To demonstrate its utility, we enhance Linux's existing support for LRU and automatic NUMA (AutoNUMA) policies [22, 29] to take advantage of kernel object tiering.

Updating LRU and AutoNUMA: Modern LRU policies track active pages and inactive pages via separate lists. Ideally, as pages become inactive, they would be migrated to slow memory, and as they become active, they are migrated to fast memory. Like prior work for two-level memories [53], we use this approach to determine which pages to migrate between memory devices. Unlike prior work, we also migrate kernel objects. Once the knodes for a file/socket becomes inactive, we immediately mark and migrate the kernel page objects they are associated with, without waiting for scans of active/inactive lists. We also enhance Linux's existing LRU policy to avoid repeated migration. We use 8-bit per-page counters to track migrations and retain such pages in fast memory. We found that less than 1% of pages met these conditions due to the shorter lifetime of kernel objects.

We also enhance AutoNUMA with KLOCs to better balance local/remote memory accesses in traditional multi-socket NUMA systems. While recent kernel patches suggest that AutoNUMA developers are considering ways of optimizing data placement in tiered memory systems, these approaches completely ignore kernel objects [29]. With AutoNUMA, the OS periodically scans a portion of a task's address space and marks the memory to force a page fault when the data is next accessed. When this address is faulted to, the data can be migrated to a memory node associated with the task accessing the memory. AutoNUMA also uses a scheduler to group tasks that share data. Baseline AutoNUMA works well for application pages but takes too long to identify kernel objects such as page cache associated with the application, corroborating results from previous work [55]. We overcome these problems by enhancing AutoNUMA with KLOCs via a simple policy: for all active KLOCs currently in use by an application, we identify related kernel objects and check if their pages are placed in local memory. We use the kmap and per-CPU lists to do this and subsequently migrate kernel objects that are remote. As we show in section 7, improving AutoNUMA with KLOCs performance by 1.4×.

5 IMPLEMENTATION

We briefly describe the components that support KLOC and then discuss our current design implications and limitations.

KLOC components: Due to the lack of multi-tiered software-controlled heterogeneous memory systems, we implement KLOCs on a dual-socket system with fast and slow memory, where slow memory is realized by throttling bandwidth. We also evaluate KLOCs on an Intel Optane system [4] to explore its benefits in an environment that requires coordinated hardware and software management. The KLOC abstraction and OS-level changes are implemented in roughly 4K lines of code, spread across different parts of Linux memory management, ext4 file system, network, and storage block driver stacks. KLOCs require no application changes except linking to a userspace shared library. The shared library approach – as opposed to a kernel configuration that enables KLOCs at the compile time – offers system administrators the option to dynamically enable and selectively control which kernel objects are included with KLOCs.

Table 2: KLOC APIs. App Dev., OS Dev., and Admin indicate KLOC API use by application-, kernel-developers, and administrators, respectively.

KLOC API	Description	API User
sys_enable_kloc()	System call to enable KLOC for an application	Admin
map_knode(knode, inode)	Map a new inode to a knode	OS dev.
knode_add_obj(knode, obj)	Add kernel object to a knode	OS dev.
itr_knode_slab(knode)	Iterate knode's kernel objects in slab tree	OS dev.
itr_knode_cache(knode)	Iterate knode's kernel objects in page cache tree	OS dev.
add_to_kmap(knode)	Add knode to global kmap	OS dev.
get_LRU_knodes(kmap)	Get LRU knodes from kmap	OS dev.
find_cpu(knode)	Find CPU that last accessed a knode	OS dev.
sys_kloc_memsize(memtype, size)	System call to limit the memory capacity use of a memory type by KLOC	Admin

KLOC usage interface: Table 2 summarizes the set of functions that we design and expose to the remainder of the kernel to manipulate KLOCs. Figure 3(c) shows an example of how to use these functions for the case where a dentry kernel object is allocated and mapped. A dentry object is used to track the hierarchy of files in a directory. The code checks to see whether there is an active knode and then performs the requisite additions to the KLOC and *KLOC map*.

KLOC memory usage: KLOCs increase memory consumption by <1% of fast memory capacity. The memory increase stems from the 8-byte red black tree pointers to cache pages and slab object structure in the *rb-cache* and *rb-slab* trees, per-CPU active and inactive lists, a linked list to track pages that need to be migrate, and other auxiliary structures. In section 7, we provide a breakdown of memory increase with KLOC. Our future work will focus on reducing these overheads.

KLOC System call cost: During a system call, the KLOCs code paths set a flag to mark an inode active and a promising candidate for allocation to fast memory. This is a fast operation. Kernel objects allocated during the system call are added to knodes. Although KLOCs use the file and network system for kernel object placement decisions, system call overheads are negligible. Kernel object migrations are asynchronous, and we use dedicated kernel threads to migrate kernel objects associated with active and inactive knodes between fast and slow memory. This can involve additional CPUs for the migration thread, but this is no different from the migration mechanisms used by modern swap managers, state-of-the-art

heterogeneous memory management systems such as Nimble [53], Thermostat [13], and others [35].

KLOC support for multi-page size. Because most Linux kernel-level objects like page cache and slab pages are allocated using 4KB pages, we mainly focus on 4KB pages. For applications that can use larger page sizes, the KLOC abstraction relies on existing Linux LRU support for active and inactive page detection and migration. Because KLOC aims to increase the placement of kernel objects to fast memory and reduce migrations, KLOCs should provide higher performance gains with THP [13, 53], although this hypothesis needs to be tested in future studies.

Table 3: We evaluate KLOCs using I/O-intensive applications that stress the storage and networking stacks.

Application	Description	Memory Footprint
RocksDB [2]	Facebook's persistent key-value store based on log-structured merge tree. We use DBbench [3] workload with 1M keys and 16 client threads. The benchmark performs 50% random and sequential writes and reads.	12.4GB
Redis [9]	In-memory key-value store that periodically checkpoints to disk. We use 16 Redis instances that serve requests from 16 clients with 4M keys with 75% sets (writes), 25% gets (reads).	14.0GB
Filebench [49]	File system benchmark using 16 threads, 13.0GB per-thread, executing 50% sequential and random reads on a 32GB file.	16.3GB
Cassandra [1]	NoSQL DB running YCSB [21] with 16 threads, 50% read-write ratio.	11.0GB
Spark [56]	Apache Spark with Hadoop, running Terrasort on 20GB of data with 16 threads. The workload first generates the dataset followed by the analytics.	32.1GB

6 EXPERIMENTAL METHODOLOGY

6.1 Evaluation Workloads

We quantify the benefits of KLOCs on the I/O-intensive workloads in Table 3. Our evaluation focuses on the Filebench, RocksDB, Redis, and Cassandra workloads because we had difficulty resolving issues brought about by the firewall settings in Spark.

6.2 Evaluation Platforms

For evaluation, we use two experimental platforms. KLOCs can be used in multiple tiered memory configurations. Evaluating KLOCs on all memory configurations is infeasible, so we focused on two extreme points – a software-managed tiered memory setup and a combined hardware/software-managed tiered memory setup. The Optane *Memory Mode* is the latter, where software is responsible for migrating data across memory nodes, but hardware is responsible for tiering data within each node. In both platforms, application and kernel object pages are managed by the OS and are transparent to the programmer.

Software-managed tiered memory. In our first platform, which we refer to as *two-tier memory*, uses the OS to control data management between a high-bandwidth, low-capacity first DRAM tier and a lower-bandwidth, higher-capacity second DRAM tier. While we would prefer using a real-world platform for these studies, there are no commercially-available tiered memory systems with entirely OS-controlled data movement, although they are expected to become viable and widely-used in the near future (e.g., die-stacked memories, disaggregated memories, etc. [41, 53]). Instead, like recent work, we leverage a two-socket system for our studies [13, 33, 41, 53]. We use thermal throttling to reduce the DRAM bandwidth in one of the sockets in a configurable manner, mimicking the activity of slower memory. Table 4 shows that fast memory is configured to 8GB of capacity at 30GB/s. This matches the raw capacity and bandwidth ranges as well as relative ratios between fast and slow memory from recent studies [17, 33, 41, 53]. We also evaluate performance for variations of fast memory capacity and slow memory bandwidth. We turn off AutoNUMA for the two-tier memory system, like recent work on Nimble [53]. This is because AutoNUMA moves pages across homogeneous NUMA nodes based on CPU affinity/locality, unlike KLOCs and Nimble, whose goal is to enable tiering across memory devices with differing performance characteristics.

Table 4: We use the two-tier memory and Optane Memory Mode platforms for our evaluations. KLOCs are used in both platforms by the OS, which controls data movement between memory tiers in the first platform, and memory sockets in the second platform.

Experimental Platforms	
Two-Tier Memory Platform	
Processor	2-socket Intel E5–2650v4 (Broadwell), 2.4 GHz cores, 20 cores/socket, 2 threads/core
SRAM Cache	512 KB L2, 25 MB LLC
Memory	Two 80 GB sockets, max bandwidth of 30 GB/sec
Storage	512 GB NVMe with 1.2 GB and 412 MB sequential and random access bandwidth
OS	Debian Trusty — Linux v4.17.0
Optane Memory Mode Platform	
Processor	2-socket Intel Xeon, 2.67 GHz cores, 32 cores/socket, 2 threads/core
SRAM Cache	512 KB L2, 25 MB LLC
DRAM Cache	16-GB DRAM hardware-managed L4 cache per socket
Memory	128-GB Intel DC Persistent DIMM per socket
Storage	1-TB Intel NVMe Block Storage
OS	Debian Trusty — Linux v4.17.0

Hardware/software-managed tiered memory. Our second platform uses a two-socket Intel Optane DC system representative of tiered memory systems that use a hybrid OS-hardware approach for data management. We configure the Optane DC system to operate in Intel Optane's *Memory Mode*, meaning that each socket uses its DRAM as a hardware-managed L4 cache of a slower-tier byte-addressable persistent memory [55]. Data movement between the L4 DRAM cache and persistent memory is controlled entirely in hardware, while the OS is responsible for data movement between sockets using AutoNUMA techniques. The DRAM L4 cache achieves 3-4× faster latency than persistent memory.

One might also consider using Intel Optane's *App Direct* mode for our studies. However, the goal of KLOCs is to manage OS kernel object tiering. Since programmers do not have direct access to kernel objects by design, it is not possible to tier kernel objects using the App Direct mode and therefore not possible to demonstrate the benefits of KLOCs in the App Direct mode. We show the configuration parameters of the Memory Mode platform in Table 4.

Table 5: KLOCs and alternatives evaluated in our studies. Approaches like *Nimble* represent prior state of the art research on application page migration [53], while *AutoNUMA* is the standard in modern OS kernels like Linux and FreeBSD.

Strategy	Description
	Two-Tier Memory Platform
Naive	Greedy approach that places application and kernel data in fast memory until it fills up. After becoming full, fast memory is available again when data in it is deallocated.
Nimble	Prior work using OS-controlled application tiering with parallel and concurrent page migration optimizations.
Nimble ++	Our extension of Nimble to migrate kernel object migration with parallel migration optimizations, but without implementation of the KLOC abstraction.
KLOCs	Original Nimble policies to identify hot application pages and mechanisms to accelerate application pages, KLOCs to associate hot/cold application pages with kernel objects, and parallel kernel page migration.
	Optane Memory Mode
AutoNUMA	AutoNUMA for application page migration between sockets, with L4 DRAM caches of persistent memory.
KLOCs	AutoNUMA with support to migrate kernel objects associated with application pages between sockets, with L4 DRAM caches of persistent memory.

Performance comparisons: We compare KLOCs against the memory management strategies in Table 5. We also compare against an ideal scenario where all application and kernel data is resident in fast memory (*All Fast Mem*), and the pessimistic scenario where all application and kernel data is in slow memory (*All Slow Mem*). Table 5 separates the tiering strategies for the two-tier versus Optane Memory Mode platforms.

For the two-tier memory platform, we consider a *Naive* approach that employs a greedy first-come, first-serve approach for allocating data in fast memory. Once fast memory becomes full, all allocations are directed to slow memory. Neither application nor kernel pages are migrated between memory tiers, meaning that fast memory becomes unavailable for allocation until some data in it is deallocated first. In contrast, *Nimble* is a recently-proposed data placement and migration scheme for application pages in tiered memory [53]. *Nimble* optimizes page hotness tracking and accelerates software-directed page migration via parallelization of page copy operations and concurrent multi-page migrations. We also enhance *Nimble* to support kernel objects in two ways. The first and most straightforward approach (*Nimble++*) is to extend *Nimble's* existing mechanisms and policies that identify and migrate hot kernel objects without the KLOC abstraction. While this approach does permit hot kernel objects to reside in fast memory, more practically, once kernel objects are evicted to slow memory, they rarely return to fast memory. The key problem is that *Nimble's* page hotness and migration control have higher latency than kernel objects' lifetimes. Hence, *Nimble++* offers sub-optimal performance because it cannot adapt to changes in kernel object hotness sufficiently rapidly. In contrast, *KLOCs* permits *Nimble* to more rapidly identify and migrate hot kernel objects associated with hot application pages with *Nimble's* parallel page migration optimizations. While *Nimble's* concurrent multi-page migration optimizations can be extended to kernel objects, we leave this for future work because of the engineering complexity. We show that just extending *Nimble* with kernel object support via KLOCs outperforms *Nimble* and *Nimble++*, despite the absence of concurrent multi-page kernel object migration.

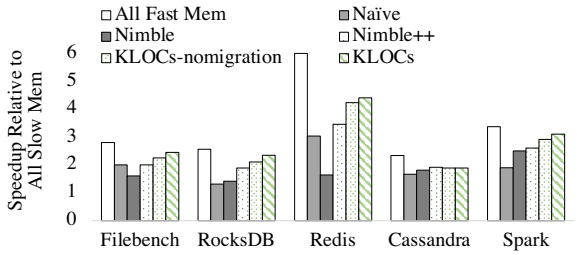


Figure 4: Performance of KLOCs on the two-tier memory platform. Speedups are relative to the *All Slow Mem* configuration. KLOCs outperform all other approaches, except for Cassandra, where they are roughly similar to *Nimble++*. *KLOCs-nomigration* shows an approach that uses KLOCs to directly allocate kernel objects to fast memory without migration. Finally, the *KLOCs* bars combine KLOCs-nomigration's direct allocation of kernel objects as well as the migration of kernel objects associated with active and inactive knodes.

For our Optane Memory Mode platform, we enhance *AutoNUMA* with KLOCs so that the OS can migrate kernel objects between sockets. Our experiments are set up such that workloads are run concurrently with another workload that streams through memory and hence interferes with our workload on one of the sockets. When interference begins to harm performance, *AutoNUMA* migrates the workload of interest to another socket where there is no interfering workload. However, while vanilla *AutoNUMA* migrates application pages, kernel object pages are ignored. This problem is resolved with KLOCs, and kernel objects are also migrated.

Table 6: Average memory increase using KLOCs in MBs compared to All Fast Mem approach.

	Filebench	RocksDB	Redis	Cassandra	Spark
Mem usage increase	44MB	101MB	83MB	12MB	43MB

7 EVALUATION

7.1 Overall Performance

Figure 4 quantifies the speedup achieved via KLOCs on the two-tier memory platform versus the alternatives in Table 4, normalized to the case when only slow memory is available. We compare *KLOCs-nomigration*, an approach that directly allocates active KLOCs to fast memory without migrating inactive kernel objects from fast to slow memory, and *KLOCs*, which also migrates kernel objects. Both approaches generally outperform other approaches. Consider, for example, filesystem-intensive workloads like RocksDB, which stores persistent key-values as a string-sorted table in hundreds of 4MB files. Because many of the files become inactive, *Naive* pollutes fast memory. *KLOCs-nomigration* directly allocates performance-critical active knode objects to available fast memory pages and achieves 1.61× throughput gains over the naive approach. However,

KLOCs-nomigration cannot move inactive kernel objects that are yet to be deallocated, reducing the available fast memory for kernel objects of an active knode. In contrast, *KLOCs* also migrates inactive kernel objects to fast memory, increasing fast memory availability, and improving performance by 1.96×. Redis, which uses only a few large files to checkpoint data, suffers cache pollution in the *Naive* case. Because Redis is also networking-intensive, the *Naive* approach is vastly outperformed by KLOCs (by 2.2×), which can ensure that socket buffers are prioritized in fast memory and can rapidly identify and migrate cold kernel socket buffers to slow memory.

In general, KLOCs also outperform *Nimble* and *Nimble++*. For Redis, KLOCs throughput increases by 2.7× over *Nimble*. Both *Nimble* and *Nimble++* leave cold kernel objects for longer in fast memory than KLOCs. *Nimble* and *Nimble++* also take longer to identify hot kernel objects and retrieve them in fast memory than KLOCs. One might initially expect that since kernel objects have short lifetimes, retrieving them into fast memory may be infrequent. However, our experiments suggest that kernel objects experience rapid phase changes in hotness, and while *Nimble* and *Nimble++* are too coarse-grained in assessing these phase changes, KLOCs can adapt to them more readily and improve performance.

Figure 4 shows that KLOCs is similar to *Nimble++* for Cassandra. This is because Cassandra uses a 512MB application-level cache for 200K keys. Because this large cache satisfies many requests at the application level, kernel I/O is reduced, performance is less sensitive to kernel object placement. Note that for the same reason, Cassandra benefits the least from the ideal case where all data is placed in fast memory. Additionally, Cassandra suffers from high Java and language overheads towards storage access combined with the use of the YCSB workload generator [21] running in a client-server configuration.

Memory Usage. Table 6 shows the increase in memory usage when using KLOCs for all applications compared to the *All Fast Mem* approach. Although KLOCs increase memory usage, the increase is $< 1\%$ of overall memory usage. For RocksDB, with the maximum memory increase (101MB), the overheads stem from metadata required for supporting KLOCs. The metadata memory increase mainly stems from 8 byte RB-tree pointer for each cache page and slab object structure that is added to *rb-cache* and *rb-slab* trees (roughly 96MB). The per-CPU active and inactive list (<800KB), a list to track pages to migrate (roughly 1MB depending on migration size), 64 byte KLOC structure attached to each open inode (<400KB), and other KLOC auxiliary bookkeeping structures also contributed to memory increase.

Hardware/software-managed tiered memory. Figure 5a shows that KLOCs outperform *AutoNUMA* in the Optane's Memory Mode configuration. The ideal scenario where all data can be maintained in local memory offers a 1.6× speedup. AutoNUMA and Nimble are able to achieve only a fraction of this improvement because they ignore kernel pages, which remain resident in the memory device where they were first allocated, even as execution shifts between NUMA nodes. KLOCs achieve performance improvements close to 1.5× over AutoNUMA and 1.4× over Nimble.

7.2 Sources of Performance Improvement

To shed light on the source of KLOCs performance improvements for the two-tier memory platform, Figure 5b quantifies the number of page cache pages and slab pages allocated in slow memory, as well as pages migrated from fast to slow memory for RocksDB. For good performance, we wish to maximize the number of pages allocated in fast memory and minimize the number of pages allocated in slow memory. Figure 5b shows that KLOCs allocate data in slow memory far less frequently than *Naive*, *Nimble*, or *Nimble++* because they are able to more quickly identify kernel objects associated with cold application pages, and migrate them to slow memory as a group. KLOCs also guard against excessive migration, which can damage performance. In particular, it requires far fewer migrations than *Nimble* and *Nimble++*. Both *Nimble* and *Nimble++* lack the knowledge of active and inactive kernel objects, and both application and kernel pages are always maintained in fast memory, hence polluting fast memory. In response, more application data migrations are necessitated between the two tiers. In contrast, KLOCs only place active kernel objects to a faster memory, reducing fast memory pollution and resulting migrations. The confluence of these factors enables KLOCs to achieve superior performance to the alternatives.

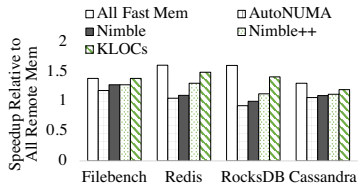
7.3 Performance Sensitivity of Kernel Objects

Different workloads are more performance-sensitive to different combinations of kernel objects. Figure 2a illustrates this, showing that all kernel objects must be included within the KLOC abstraction for maximal performance benefit. To generate these results, we incrementally add groups of kernel objects to the KLOC abstraction and quantify performance improvements. Initially, we tier just the application pages and always assign kernel objects to fast memory only. Then we incrementally add KLOC support for page caches, followed by journals, slab objects, socket buffers, and block I/O. For each of these configurations, kernel objects excluded from KLOCs are always maintained in fast memory. We find that many workloads benefit from including page cache pages within KLOCs, but other workloads like Redis also benefit from socket buffers, etc. The conclusion is that workloads employ kernel objects in diverse ways, and a truly robust KLOC abstraction must include as many kernel object types as possible.

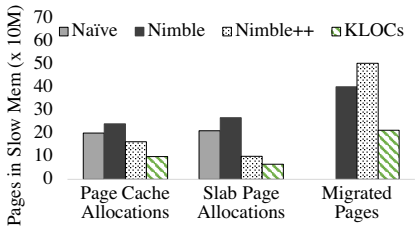
KLOCs also offer complementary benefits to existing I/O prefetching techniques in modern OSes. We elide results for brevity, but find that with *Naive*, *Nimble*, and *Nimble++*, prefetching can amplify the likelihood fast memory is polluted by cold application pages and hence, their associated kernel object pages. KLOCs can quickly identify the kernel object pages associated with cold application pages and migrate them to slow memory. This is particularly useful for not just applications where some prefetched pages are indeed useful (e.g., RocksDB, Redis, and Cassandra), but also for workloads with more random access patterns (e.g., Filebench) where pollution from prefetching is pernicious. For instance, augmenting prefetchers with KLOCs improves RocksDB throughput by 1.26×.

7.4 Sensitivity to Bandwidth and Capacity

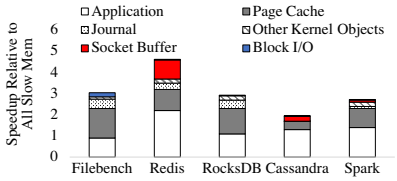
Finally, Figure 6 quantifies KLOC speedup as a function of the relative memory bandwidth of fast and slow tiers of memory and



(a) KLOCs speedup versus vanilla AutoNUMA and an ideal case where all accesses are to local memory, all normalized to the case where all accesses are to remote memory.



(b) The number of pages allocated to the page cache, slab allocations, and the number of pages migrated from slow to fast memory for RocksDB in two-tier memory.



(c) Contribution of different kernel object types to the performance of KLOCs on the two-tier memory platform. All kernel objects excluded from KLOCs are placed in fast memory.

Figure 5: Figure 5a shows the speedup offered by KLOCs in the Optane platform's Memory Mode over the worst-case configuration where all data is serviced from remote memory, the vanilla AutoNuma and the Nimble configurations. Figure 5b shows the number of pages allocated in slow memory for page cache objects and slab objects (in units of 10 million pages), and pages migrated from fast to slow memory on the two-tier memory platform for RocksDB. A good approach is to maximize the number of pages allocated in fast memory (therefore, we wish to minimize page cache page and slab page allocations in slow memory) and migrate as many cold pages to slow memory as possible. KLOCs does both better than *Nimble* or *Nimble++*. Figure 2a shows the impact of different kernel objects on the KLOCs performance.

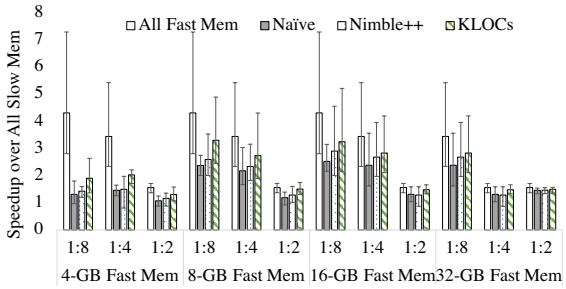


Figure 6: Speedup on a two-tier memory platform. Fast memory capacity is varied from 4GB to 8GB to 32GB. Bandwidth differentials between slow and fast memory are varied from the scenario where fast memory has 8× more bandwidth than slow memory (1:8 x-axis label) to 4× to 2×. All bars show the average across all workloads, and variance bars capture the maximum and minimum speedup achieved across workloads.

the capacity of fast memory in the two-tier memory platform. We show average speedups across all workloads, with variance bars indicating the minimum and maximum speedups achieved by our workload per configuration. KLOCs offer superior performance across all bandwidth and capacity configurations, particularly as the bandwidth differential between fast and slow memory becomes more pronounced. KLOCs offer more speedup as capacity becomes progressively limited, and as the memory bandwidth differential grows. In general, the speedup benefits over *Nimble* and *Nimble++* remain consistent and peak for mid-scale fast memory capacities of 8GB, especially for higher bandwidth differentials. As fast memory capacity increases, slow memory is used less often, reducing the performance difference of all tiering approaches.

8 CONCLUSION

We present KLOCs, a new OS abstraction to systematically group and manage kernel objects data. As hardware vendors increase system memory capacities, the total memory footprint of kernel objects has grown to the extent that it can no longer be treated as an afterthought, particularly for heterogeneous memory systems. Many refinements and enhancements can be made to this initial KLOC abstraction, but by showing that even a proof-of-concept prototype can achieve 1.4×-2.7× higher throughput than traditional systems, we highlight the exigent need to investigate better kernel object management in the systems research community.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and Yungang Bao (our shepherd) for their insightful comments and feedback. We thank the members of the Rutgers Systems Lab for their valuable input. This material was supported by funding from NSF grant CNS-1910593, NSF CAREER #1916817, and experimental platforms funded by NSF grant CNS-1730043. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF.

REFERENCES

[1] [n. d.]. Apache Cassandra. http://cassandra.apache.org/.
[2] [n. d.]. Facebook RocksDB. http://rocksdb.org/.
[3] [n. d.]. Google LevelDB . http://tinyurl.com/osqd7c8.
[4] [n. d.]. Intel-Micron Memory 3D XPoint. http://intel.ly/1eICR0a.
[5] [n. d.]. Intel VTune Amplifier. https://hpc.llnl.gov/software/development-environment-software/intel-vtune-amplifier.
[6] [n. d.]. Knights Landing (KNL): 2nd Generation Intel® Xeon Phi™ Processor. https://www.alcf.anl.gov/files/HC27.25.710-Knights-Landing-Sodani-Intel.pdf.
[7] [n. d.]. Linux Page Migration. https://www.kernel.org/doc/Documentation/vm/page_migration.
[8] [n. d.]. Nginx memory usage. https://www.nginx.com/blog/nginx-websockets-performance/.
[9] [n. d.]. Redis. http://redis.io/.
[10] [n. d.]. VMWare vNUMA. https://www.vmware.com/files/pdf/techpaper/VMware-vSphere-CPU-Sched-Perf.pdf.
[11] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: Transparently Self-Replicating Page-Tables for

Large-Memory Machines. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*.

[12] Inc Advanced Micro Devices. [n. d.]. AMD High Bandwidth Memory. https://www.amd.com/en/technologies/hbm.

[13] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 631–644. https://doi.org/10.1145/3037697.3037706

[14] Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. 2011. 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)*. Portland, OR.

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

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

[17] Yu Chen, Ivy Peng, Zheng Peng, Liu Xu, and Bin Ren. [n. d.]. ATMem: adaptive data placement in graph applications on heterogeneous memories. *Code Generation and Optimization (CGO)* ([n. d.]).

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

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

[20] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable Address Spaces Using RCU Balanced Trees *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 199–210. https://doi.org/10.1145/2150976.2150998

[21] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. Indianapolis, Indiana, USA.

[22] Jonathan Corbet. [n. d.]. AutoNUMA: the other approach to NUMA scheduling. https://lwn.net/Articles/803663/.

[23] Jonathan Corbet. [n. d.]. Generic red-black trees. https://lwn.net/Articles/500355/.

[24] Jonathan Corbet. [n. d.]. Linux Per-CPU Lists. https://lwn.net/Articles/258238/.

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

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

[27] Vishal Gupta, Min Lee, and Karsten Schwan. 2015. HeteroVisor: Exploiting Resource Heterogeneity to Enhance the Elasticity of Cloud Platforms. *SIGPLAN Not.* 50, 7 (March 2015), 79–92. https://doi.org/10.1145/2817817.2731191

[28] Anthony Gutierrez, Michael Cieslak, Bharan Giridhar, Ronald G. Dreslinski, Luis Ceze, and Trevor Mudge. 2014. Integrated 3D-stacked Server Designs for Increasing Physical Density of Key-value Stores. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. Salt Lake City, Utah, USA.

[29] Ying Huang. [n. d.]. autonuma: Optimize memory placement in memory tiering system. https://lwn.net/Articles/803663/.

[30] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/2485922.2485957

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

[32] Crobett Jonathan. [n. d.]. Linux Radix Trees. https://lwn.net/Articles/175432/.

[33] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 521–534. https://doi.org/10.1145/3079856.3080245

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

[35] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/3297858.3304053

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

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

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

[39] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (Feb. 1991), 21–65. https://doi.org/10.1145/103727.103729

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

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

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

[43] Moinuddin K. Qureshi and Gabe H. Loh. 2012. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 235–246. https://doi.org/10.1109/MICRO.2012.30

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

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

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

[47] Jia Rao, Kun Wang, Xiaobo Zhou, and Cheng zhong Xu. 2013. Optimizing virtual machine scheduling in NUMA multicore systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. 306–317. https://doi.org/10.1109/HPCA.2013.6522328

[48] Samsung. 2020. Samsung X-Cube. https://news.samsung.com/global/tag/samsung-x-cube.

[49] Tarasov Vasily. [n. d.]. Filebench. https://github.com/filebench/filebench.

[50] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 347–362. https://doi.org/10.1145/3314221.3314650

[51] Wu, Fengguang and Xi, Hongsheng and Li, Jun and Zou, Nanhai. 2019. Linux readahead: less tricks for more. (08 2019).

[52] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16)*.

[53] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 331–345. https://doi.org/10.1145/3297858.3304024

[54] Zi Yan, Jan Vesely, Guilherme Cox, and Abhishek Bhattacharjee. 2017. Hardware Translation Coherence for Virtualized Systems. In *International Symposium on Computer Architecture (ISCA '17)*.

[55] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2019. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. arXiv:cs.DC/1908.03583

[56] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113

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