VMem: Enabling Application Management of Physical Memory Resources

J. Zach McMichael
jmcmicha@vols.utk.edu
University of Tennessee
Knoxville, Tennessee, USA

Jacob Malloy
jmalloy1@vols.utk.edu
University of Tennessee
Knoxville, Tennessee, USA

Brandon Kammerdiener
brandon.kammerdiener@intel.com
Intel Corporation
Santa Clara, California, USA

Michael R. Jantz
mrjantz@utk.edu
University of Tennessee
Knoxville, Tennessee, USA

Terry Jones
trjones@utk.edu
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

Kshitij A. Doshi
kshitijd@uber.com
Uber Technologies
San Francisco, California, USA

Abstract

Computing trends are leading high performance and enterprise platforms to adopt memory systems with increasingly complex architectures. As a result, many high end systems now include multiple types of memory with different capabilities and performance or distributed memory resources connected via a fast interconnect. New data management strategies are needed to exploit the unique advantages of these diverse and distributed architectures. As the primary generators of memory accesses, applications are well-suited to guide and tailor memory management for optimizing usage of these architectures. However, conventional data management in the operating system often proceeds with little knowledge of application intents or behaviors. This semantic gap limits optimization opportunities and can lead to inefficient utilization of complex memory resources.

To address these challenges, this work proposes VMem: an application runtime and programming interface for enabling direct application control of physical memory resources. Designed and developed in Linux, VMem leverages standard Linux features and system calls to delegate key physical memory management tasks, including the allocation and recycling of physical memory, to the application itself. VMem does not require custom kernel code or non-standard hardware, and through integration with the memory allocator, can be deployed for use with many applications without needing to update or recompile application source. Experiments with the SPEC® CPU 2017 benchmarks, this work demonstrate that many applications can use VMem to exert control over physical memory resources with little or no overhead compared to the default software stack. Additionally, this work discusses opportunities to improve memory utilization with VMem and demonstrate this potential by using it to implement an optimization that uses page replication to reduce costs associated with data migration.

CCS Concepts

• **Software and its engineering** → *Runtime environments*; • **Computer systems organization** → *Heterogeneous (hybrid) systems.*

Keywords

Memory management, runtime systems, heterogeneous memory systems

1 Introduction

As computing advances are increasingly relying on data-driven analyses, including artificial intelligence (AI) and machine learning (ML), demands for high-throughput, low-latency processing of larger and larger sets of data in memory are continuing to rise. At the same time, the need for high-density sharing has led to the widespread adoption of machine configurations with large amounts of memory attached to distributed nodes and connected through efficient networking resources. New media technologies, such as high bandwidth memories, power-efficient and non-volatile RAMs, and many other new accelerator options, including processing-in-memory (PIM), and new memory interconnect options, including the Compute Express Link (CXL), are bringing rich opportunities to address the diverse needs of modern applications under various cost, performance, and power constraints. In response to these trends, most cluster and warehouse-scale computing systems now include a heterogeneous mix of memory devices and organizations designed to enable the combined benefits of their unique capabilities and support the diverse and multi-tenant workloads deployed in modern data centers and supercomputers.

As a result, new data management strategies are needed to capitalize on the different strengths of diverse and distributed memories. Specifically, the system must be able to efficiently match application data to the type of memory that best suits its purpose for the optimal amount of time. Applications, as the primary generators of memory usage, are well-suited to guide such tasks. However, conventional data management approaches are limited in how they use application-level information due to some divisions that have traditionally been present during memory management. While applications control the structure and usage of program data, many data management tasks including: the allocation and recycling of physical memory, updates to application page tables, translations of virtual to physical addresses, and swapping of volatile to persistent storage; are under the purview of the operating system (OS) and hardware. Therefore, these tasks proceed with little or no knowledge of application behavior. *This semantic gap limits optimization opportunities and can lead to inefficiencies in how application data is mapped to diverse and distributed memory architectures.*

To address these shortcomings, this work proposes a new memory management framework, called VMem, that aims to reduce the semantic gap between application-level data usage and system-level memory management. VMem acquires and organizes physical

memory into logical pools corresponding to divisions in the underlying memory architecture. It then delegates key operations on physical memory, including page fault handling and recycling, to the application itself, thereby empowering the application to map, use, and manipulate these resources for their own program data. In this way, VMem provides an environment for researchers and engineers to design and implement system-level memory management strategies and optimizations in tight coordination with application semantics and behavior.

To implement this functionality, VMem employs system calls and facilities that are standard on modern Linux platforms. As a result, VMem does not require any kernel modifications or non-standard hardware. Moreover, its novel design supports multi-process and multi-application usage scenarios, and in many cases, applications can leverage features and optimizations enabled by VMem automatically, and without updating program source code, by dynamically linking a custom allocator that invokes the VMem API.

This work makes the following important contributions:

- It describes the design and implementation of VMem: a user-level runtime and API that enables applications to direct and fine-grain control over physical memory resources. The VMem framework has been developed open source and will be available for download upon publication of this work.
- It evaluates the execution time overhead and limitations of this approach. Specifically, it characterizes the key operational costs of VMem with different page sizes on server class Intel® hardware. Through experiments with the standard SPEC® CPU 2017 benchmark suite, it shows that many applications can use VMem with little or no execution time overhead compared to the default Linux memory manager.
- To demonstrate the potential of this approach to improve efficiency on complex memory platforms, it uses VMem to implement an optimization that leverages *page replication* to reduce costs associated with migrating data from one device to another. The evaluation shows that this optimization can potentially increase migration throughput by multiple orders of magnitude (20× to over 295×) for applications that need to move memory regions with infrequent writes.

2 Background and Related Work

Software-Directed Data Tiering: As computing trends have led to higher demands on an increasingly complex data path, many systems have begun to employ *software-directed tiering* of data in memory to enable more flexible and effective utilization of diverse memory hardware. Software-directed tiering empowers the OS, sometimes with feedback from or in coordination with application software, to assign data into different memory tiers and migrate data between tiers as needed. Modern implementations of this approach are similar to data management on non-uniform memory architecture (NUMA) platforms [17], with each type or tier of memory represented as its own NUMA domain. In many cases, the system also exposes data placement controls to user-level programs through the system call interface. For example, Linux applications can use the `mbind` or `move_pages` system calls to request or require that a specific range of virtual memory be backed with physical pages from a particular memory tier. These finer-grained controls enable applications to coordinate tier assignments with allocation and usage patterns, potentially enabling powerful efficiencies.

Despite its advantages, software-directed data tiering is often underutilized due to some limitations of current approaches. First, software-directed data migration often incurs high execution time costs due to the need to align virtual and physical memory addresses. Thus, these approaches are less adaptive than hardware-based alternatives (e.g., caching). Some prior works have proposed system-level techniques to reduce migration costs in NUMA or tiered memory platforms. For example, Nimble [33] proposed a range of optimizations, including native support for transparent huge page migration and multi-threaded page migrations, to reduce migration costs on heterogeneous memory platforms. The Carrefour algorithm [7] includes a feature that periodically (and concurrently) replicates memory pages on different NUMA nodes. In this way, the system can quickly "migrate" pages with an up-to-date replica on the target node by only updating the application page tables (i.e., additional data copies are often not necessary). While these optimizations significantly improve migration throughput, they are not accessible on many enterprise and scientific computing platforms because their implementations are only available as patches to specific kernel versions. By delegating control of the application page tables to a user-level runtime, VMem can enable *portable* implementations of most memory migration optimizations. Indeed, Section 6 of this work demonstrates and evaluates this capability with an in-memory replication optimization implemented entirely in the VMem runtime.

Another significant hurdle for software-directed data tiering is that the system-level routines that assign and move program data to different memory tiers require accurate and timely knowledge of memory usage patterns in order to be effective. In many cases, applications can provide this information directly through current interfaces, but doing so requires expert knowledge and modifications (and recompilation) of program source.

Researchers have recently proposed a variety of tools and techniques that seek to address this limitation by automating all or part of the process of understanding how applications use memory and using this information to guide memory tiering. For example, several prior works have employed profiling of architectural divisions, such as pages or cache lines, and use system-level heuristics to steer data with recent use or frequent reuse into the faster, but smaller, memory tiers [1, 6, 16, 31]. While these approaches are completely transparent to application software, they are still limited by the semantic gap between systems and applications because they proceed without knowledge of logical data structures or application intents. Some other works have attempted to bridge this gap by integrating architectural profiling with compiler or runtime instrumentation and then using this information to steer memory allocation and tiering within both the application-level allocator and system-level memory manager [2, 8, 15, 19, 23, 26, 29, 32]. While our present work does not propose or evaluate any software-directed data tiering strategies within VMem, its design facilitates further integration of application and system-level memory management activities. Specifically, by providing a common runtime where applications can control the layout of objects in the virtual address space, collect information regarding their own data usage, and also directly control how their data are mapped to physical memory resources,

VMem can potentially improve the efficiency and portability of these existing solutions.

Microkernels and Exokernels: Systems researchers have long recognized that the semantic gap between application behavior and system-level management limits customization opportunities and can lead to inefficient utilization of system resources. Indeed, performance is a key motivation for many alternative kernel designs, including microkernels [9, 21, 22] and exokernels [10], that provide applications with more direct access to memory resources. While some of these projects, such as Dune [4] and ExtMem [13], aim to limit OS modifications and preserve the application-system interface, all of them require custom kernel patches or modules, which can prevent their adoption for many applications and platforms. In contrast, VMem is built upon features that are now standard on modern Linux platforms and does not require custom operating system code or even updates to application software, in many cases.

Other Works Leveraging Userfaultfd: This work is most closely related to other recent efforts that have relied on the Linux `userfaultfd` facility to delegate some aspects of physical memory management to application software. The UMap project leverages `userfaultfd` to enable applications to customize caching, prefetching, and eviction policies for different applications and backing stores [25]. In contrast to VMem, it does not support data tiering or management of anonymous memory regions. HeMem employs `userfaultfd` to control management of application data across pools of physical memory corresponding to distinct memory device tiers [28]. However, the entire HeMem runtime links directly into a single application process and does not support multi-process data management. Additionally, HeMem relies on an architecture specific filesystem (DAX) to allocate persistent memory resources, and thus, requires some minor kernel modifications to control these resources with `userfaultfd`. Conversely, VMem leverages a separate process (i.e., the VMem Server) to enable user-level memory management for multiple concurrent processes and does not require any kernel modifications because it allocates memory for each device using the standard NUMA interface. Moreover, while VMem also aims to enhance data tiering, its runtime and API are primarily designed to enhance application control of physical memory, and thus, enable a broader set of use cases and optimizations than HeMem.

3 The VMem Framework: Enabling Application Control of Physical Memory

Figure 1 depicts the main components of our VMem framework. It primarily consists of two new pieces of software:

(1) The VMem Server acquires free memory resources from the different types of memory that are present on the platform and organizes these pages into shared memory pools. These pools are then used to serve the memory needs of connected VMem processes.
(2) The VMem Runtime is a lightweight runtime that connects an application process to the VMem server, thereby enabling it to participate in shared memory management with VMem. It implements an application programming interface (API) that allows applications to register virtual address ranges with VMem and control allocation and recycling of the physical memory resources exposed by the VMem server within these ranges.

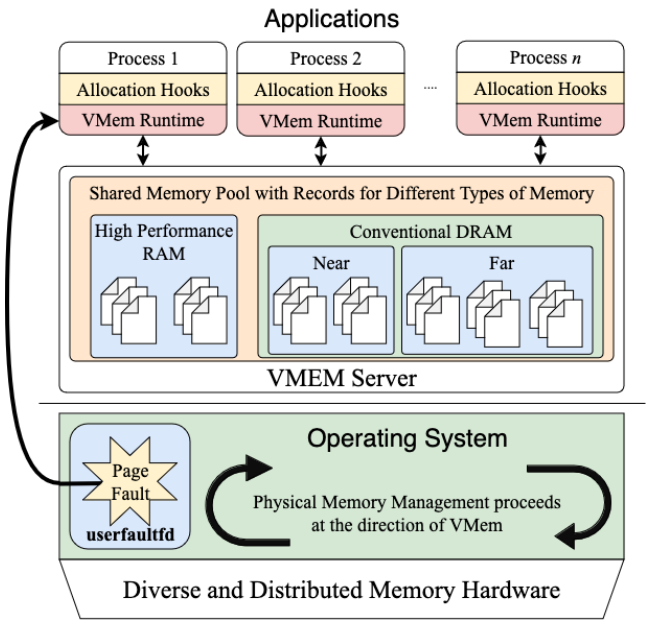


Figure 1: Design overview of the VMem framework.

The VMem framework is implemented on the Linux platform and does not require any kernel modifications or non-standard configuration options to enable fine-grained memory management features for participating applications. Rather, it leverages standard Linux system calls, including `memfd_create`, `mmap`, and `userfaultfd`, to implement its core functionality. Let us next describe how these facilities are used to implement VMem.

3.1 The VMem Server

The VMem Server provides a holistic view of the available memory resources and exposes these resources to VMem Runtime processes through a set of shared memory pools. To initialize the shared memory pools, the VMem server creates an anonymous file corresponding to the physical memory tiers that are present on the platform. It then maps each anonymous file into its address space as shared memory and populates the shared mappings with physical memory corresponding to the appropriate tier of memory. For these operations, the current implementation of VMem employs several standard Linux system calls, including: `memfd_create` to create the anonymous files, `mmap` to map these files as shared memory, and `mbind` to ensure that the shared ranges are populated with physical memory corresponding to the appropriate tier of memory. In this way, VMem can create a distinct pool of shared memory for each device tier that is distinguished as its own NUMA node on the underlying platform. However, it is important to note that this design is not limited to standard NUMA divisions, and can enable applications to manage physical memory resources corresponding to other architectural characteristics, such as DRAM banks and row buffers, as discussed in Section 7.

Function Name	Return Value	Arguments
`register_vregion`	`int` indicating success or failure	`void *addr`, `size_t len`, `int prot` `int flags`, `int fd`, `off_t offset`
`release_vregion`	`int` indicating success or failure	`void *addr`, `size_t len`

Table 1: API for registering and unregistering virtual regions in VMem.

Next, the shared ranges are subdivided into pages, called *vpages*, which are then inserted into free lists corresponding to the architectural characteristics of the underlying memory hardware. The vpages and their associated lists are created and mapped into the VMem Server as shared memory, which allows other processes that link the VMem runtime to access and manipulate them directly. In addition to metadata for their associated data structures, each vpage maintains information about the anonymous file and offset from which it was originally mapped. In this way, a VMem process can allocate physical memory corresponding to a particular type of memory by selecting a vpage from the appropriate list and mapping its associated file and offset into its virtual address space. Moreover, VMem enables applications to coordinate vpage allocation and recycling with system-level events and activities, including page faults, using the `userfaultfd` facility, as described next.

3.2 The VMem Runtime

Along with the VMem Server, VMem includes a lightweight application runtime, called the VMem Runtime, which is implemented and built as a shared library (`.so`) file. To use VMem, applications dynamically link the VMem Runtime into their address space prior to invoking their main execution routine (e.g., by using the `LD_PRELOAD` facility in Linux). During initialization, the VMem Runtime connects to the (previously initialized) VMem Server running on the same platform and collects information regarding the shared memory pools and available vpages. The application can then access, manipulate, and map the shared resources into its own address space by invoking facilities provided by the VMem Runtime.

To coordinate application-directed memory management with system-level events, the VMem Runtime employs the `userfaultfd` facility in Linux. Introduced in Linux v. 4.3, `userfaultfd` enables the OS to delegate the handling of page fault events to user processes. To use this feature, the application must first register the virtual address ranges for which it wishes to control page fault handling with the OS. When the OS receives a fault within a registered range, rather than attempt to satisfy the fault itself, the OS inserts an event describing the fault into a message queue that is shared with the process that registered the range. Meanwhile, the VMem Runtime attached to the application process has a separate thread that polls this message queue and responds to each fault event as it receives them. In this way, the VMem Runtime can map virtual ranges to physical memory resources corresponding to vpages in the VMem Server. Furthermore, by satisfying faults with vpages corresponding to the appropriate device or architectural division, this design also enables application software to control how logical program data maps to specific hardware resources.

3.3 Managing Application Memory with VMem

The VMem Runtime includes several facilities for applications to register and manage their own heap data *automatically* and without requiring any updates to or recompilation of program code. Specifically, applications that link the VMem Runtime may also optionally link our custom memory allocator, called bkmalloc [3]. bkmalloc is a general-purpose malloc implementation with similar capabilities as other allocators, such as the GNU allocator or jemalloc, but it includes additional features to control and optimize memory management for certain usage scenarios. For this work, we leverage a bkmalloc feature that enables applications to install callback routines on allocator events, including `mmap` and `munmap`, which expand and contract the application's virtual address space. For these events, we install a callback that registers all private anonymous virtual ranges with VMem, thereby delegating faults within these ranges to the VMem Runtime.[1]

To facilitate the use and management of the virtual regions registered with VMem, the VMem Runtime maintains its own internal data structures known as *vregions*. Along with address and size information, the vregions include information regarding permissions and policies that may be useful for managing the data within each region. Thus, the vregion structure is analogous to the `vm_area_struct` that is used in the Linux kernel memory manager. Similarly, the VMem Runtime maintains its own internal page table, known as the *virtual page table* (or *vpt*)), that maps virtual pages in the application to the shared set of vpages.

Table 1 presents the VMem API that the framework uses to register and unregister virtual regions with VMem. Note that the interface for registering and unregistering vregions is similar to the `mmap` and `munmap` routines in Linux. In this way, the vregions capture the relevant protections, visibility flags, and file information that are needed to satisfy faults in the corresponding address range. This design also allows applications to update an existing vregion with additional information, such as NUMA preferences or management policies, in a similar manner as `mbind` or `mprotect` updates the `vm_area_struct` in Linux (e.g., consider the `vmem_mbind_vregion` function described in Section 6). Moreover, by lifting the management of physical resources from kernel to user space, VMem provides applications with portable mechanisms to share a broad range of information for guiding memory management.

Now, when an application faults on an address that has been registered with VMem, the host OS will catch the fault and forward the faulting event to the application through the `userfaultfd`

[1]While the implementation presented in this work employs bkmalloc callbacks to register application data ranges with VMem, VMem does not strictly depend on bkmalloc to implement its core functionalities. Applications that use the default system allocator or an alternative allocator can still leverage VMem features by invoking its API directly from the runtime or application itself.

message queue. A dedicated thread within the VMem Runtime will then process the event and determine the virtual page associated with the faulting address. Using the page address, the runtime will lookup the associated vregion, which may contain file or policy information necessary to complete the fault. Next, the runtime will query the shared free lists to find a free vpage that can be used to satisfy the fault. If a suitable vpage is found, the runtime will then map the in-memory file and offset associated with the selected vpage into the virtual addresses corresponding to the page fault using the `MAP_FIXED` flag with the `mmap` system call. In this way, subsequent accesses to these virtual addresses will resolve to the physical memory resources backing the selected vpage.

3.4 Limitations & Risks

3.4.1 Support for File-backed and Shared Memory Regions. Since the current VMem implementation only supports private anonymous virtual regions (i.e., regions mapped with the `MAP_ANON` and `MAP_PRIVATE` flags set), it is primarily used to manage anonymous allocations on the application heap. However, the design can potentially be extended to support file-backed and shared memory mappings. For example, to handle faults for private file-backed memory, one option is to remap the faulting address to vpage memory as described above and then read the appropriate file contents into the vpage memory buffer prior to returning from the fault. While this approach requires an additional copy from kernel file buffers to the vpage memory buffers, it enables VMem to control and manage both file-backed and anonymous private mappings in a consistent way. Alternatively, VMem could relinquish some control of non-anonymous mappings to the operating system and attempt to manage trade-offs between fault handling overheads and optimization opportunities.

Support for shared memory mappings requires some additional capabilities and extensions to VMem. Since the vpage memory buffers are also shared, these buffers can also be used to support memory mappings that are shared among separate processes. However, new vregion fields and data structures are needed to keep track of how shared regions are backed by the filesystem. Moreover, since updates must be carried through to the underlying file, VMem would need to coordinate with the operating system to implement and enforce these updates. While support for shared memory mappings can potentially extend the benefits of VMem to a broader range of applications and usage scenarios, the primary goal of this work is to present the essential design elements of VMem and demonstrate its potential for enabling application control and management of physical memory resources. Hence, we leave support for shared memory mappings in VMem as future work.

3.4.2 Isolation for Multi-Process and Multi-Tenant Execution Scenarios. The VMem framework supports the management and usage of shared hardware resources by mapping and exposing a subset of physical memory on the host platform as a shared memory device. This design choice reflects the primary objectives of VMem, which are to enable more effective memory management on complex architectures through increased flexibility and tighter coordination among tasks that participate in memory usage and management. For execution scenarios where VMem connects a set of trusted and well-behaved (but otherwise unrelated) processes, VMem does not impose new requirements or change how memory is viewed by application software. Operations that map and manage shared memory resources are always encapsulated in the VMem runtime, which coordinates with the VMem server and other connected processes to ensure that any shared resources that it allocates will not be accessible in any other connected processes.

However, in scenarios where an adversarial or compromised application connects to the VMem server, this application could potentially gain access to data mapped into the address space of other connected processes. Specifically, since the VMem server shares vpage table and vpage data with each connected VMem runtime, the isolation of memory resources that is typically enforced by privileged execution in the operating system can no longer be guaranteed among connected VMem processes. At this stage, we view this additional vulnerability as an acceptable tradeoff that is not in conflict with our primary research goals. VMem can potentially enable customized and more effective memory management for multi-process execution scenarios as long as each process in the group is trusted or accesses only non-sensitive data. As we take this work forward, we plan to investigate techniques to prevent sharing data among processes connected to the same VMem server, while still enabling customized memory management for multi-process execution scenarios.

3.4.3 Memory Paging and Management Overheads. While delegating page management to VMem can enable much greater flexibility and control over how applications manage their own physical memory resources, this design's reliance on user space polling of event queues can introduce significant overheads into the page fault path. Each fault now requires at least two new transfers between the system and application: one to invoke the fault handler in the VMem Runtime and another that invokes the system to map application addresses to the appropriate vpage file and offset. There are additional costs to align VMem management with physical page management in the OS, including: additional creation, splitting, and merging of `vm_area` (VMA) structures as well as increased contention on VMA locks such as the `mmap_lock`.

For applications with frequent faults, and especially multi-threaded applications, these additional costs can lead to significant execution time overheads (e.g., see Figure 2b in Section 5.2). However, the flexibility of our design can mitigate these costs for many applications and usage scenarios. Specifically, VMem does not require the vpage size to be equal to the host platform page size, but rather, can optionally set the vpage size to be a multiple of the host page size. Increasing the vpage size reduces the number of faults managed by VMem, and thus reduces paging and management overheads. This feature does not eliminate all the overheads associated with user level page fault handling, and in fact, may introduce other costs in certain cases (e.g., increased capacity or worse cache utilization). *However, for many applications and usage scenarios, increasing the vpage size restricts VMem overheads to the point where there is little or no noticeable impact on application performance, even if the workload under VMem control is running at full system scale. For these cases, there is significant potential to improve system performance and efficiency by leveraging optimizations and data management strategies enabled by VMem.*

Configuration	Description	Write (inc. fault)		Read (no fault)	
		per-page (μs)	TP (GB/s)	per-page (μs)	TP (GB/s)
default-4KB	Default Linux memory management software stack (kernel v. 6.14.5) with 4 KB pages.	1.62	2.37	0.32	12.09
vmem-4KB-4KB	The current implementation of VMem with 4 KB vpages and 4 KB pages in the host Linux OS.	6.09	0.63	0.3	12.79
default-2MB	Default Linux memory management software stack with 2 MB pages (using hugetlbfs [18]).	482.45	4.07	159.71	12.28
vmem-2MB-4KB	The current implementation of VMem with 2 MB vpages and 4 KB pages in the host Linux OS.	600.13	3.25	145.6	13.42
vmem-2MB-2MB	The current implementation of VMem with 2 MB vpages and 2 MB pages in the host Linux OS.	452.25	4.33	166.98	11.7

Table 2: Microbenchmark performance with VMem. The write operations include time for a 4 KB or 2 MB page fault (depending on the page size), while the read operations show performance after the data has already been faulted in. For the *vmem-2MB-4KB* configuration, the table uses the vpage size of 2MB to calculate per-page performance.

4 Experimental Setup

Before proceeding to the evaluation of the VMem framework, let us describe our experimental platform and methodology.

Platform Details: We built and deployed VMem on an Intel® Linux platform (kernel v. 6.14.5) with two Xeon® CPU Max 9468 processors (code named Sapphire Rapids or SPR). Each processor includes 48 physical compute cores, each with 2.1 GHz base and 3.5 GHz max clock speeds, as well as a 105 MiB shared L3 cache. Intel® Hyper-Threading is enabled. Hence, each processor presents 96 logical computing cores to the operating system. Each node is also subdivided into four sub-NUMA clusters (SNCs), where each SNC contains 12 physical computing cores (or 24 logical cores), a slice of the L2 and L3 cache, and their own integrated memory controller. Similar to conventional NUMA devices, applications can control access to the cores and memory on each SNC through the Linux NUMA API. The memory system on each node includes eight 32 GB, 4800 MT/s, SK Hynix DDR5 SDRAM memory modules organized in a dual channel configuration. Hence, the system has a total of 512 GB of DRAM across the entire platform. Each computing node accesses memory on the remote node through an Intel® Ultra Path Interconnect (UPI) link that supports transfers up to 16 GT/s.

The machine also includes a set of Data Streaming Accelerator (DSA) devices that can accelerate data movement among local and remote memory devices. For our experiments that use the DSA in Section 6, the system is configured so that regions that are being copied from one node to another are divided into a set of batches of equal size and then transferred simultaneously using the eight DSA engines on the machine.[2] While both processors are also integrated with 64 GB of High Bandwidth Memory (HBM) and the system is configured in Flat (i.e., software-managed access) Mode, the HBM devices are unused in this study. Although VMem has promise to enable new optimizations and efficiencies for heterogeneous memory scenarios, this work focuses on the basic operational overheads and potential for reducing data migration costs with VMem, and thus, our experiments use only local and remote DDR5 SDRAM to simplify presentation of results.

[2]Each batch always contains up to 512 vpages, but the vpage size varies from 4 KB to 2 MB in our experiments.

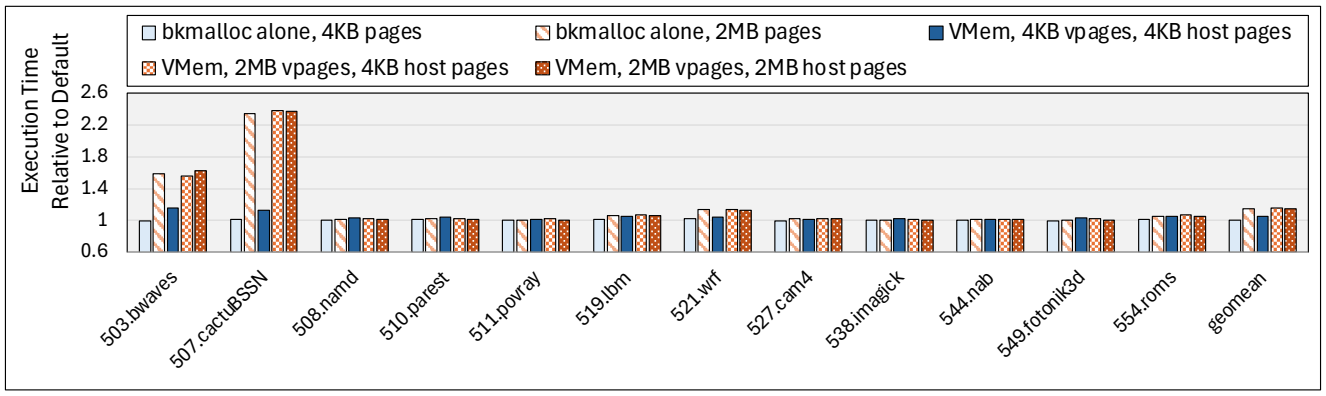
Methodology: Each experimental run initializes a single instance of the VMem Server with separate pools of memory on each server node. For this initial study, we augmented VMem with a simple free-list allocator that satisfies faults with free vpages in a first-in, first-out (FIFO) fashion. Unless a vregion policy indicates otherwise, the VMem server will service faults from the local DRAM pool if free vpages exist, and otherwise, from the remote pool. For the experiments with benchmark applications, we configure the application process to link the bkmalloc allocator prior to invoking its main routine using the `LD_PRELOAD` environment variable. During this initialization process, bkmalloc also links and invokes the VMem Runtime to connect to the running VMem Server. Aside from the VMem Server and application processes, all experimental runs execute on an otherwise unoccupied machine. All results present the mean average of five experimental runs.

5 VMem Overheads and Performance

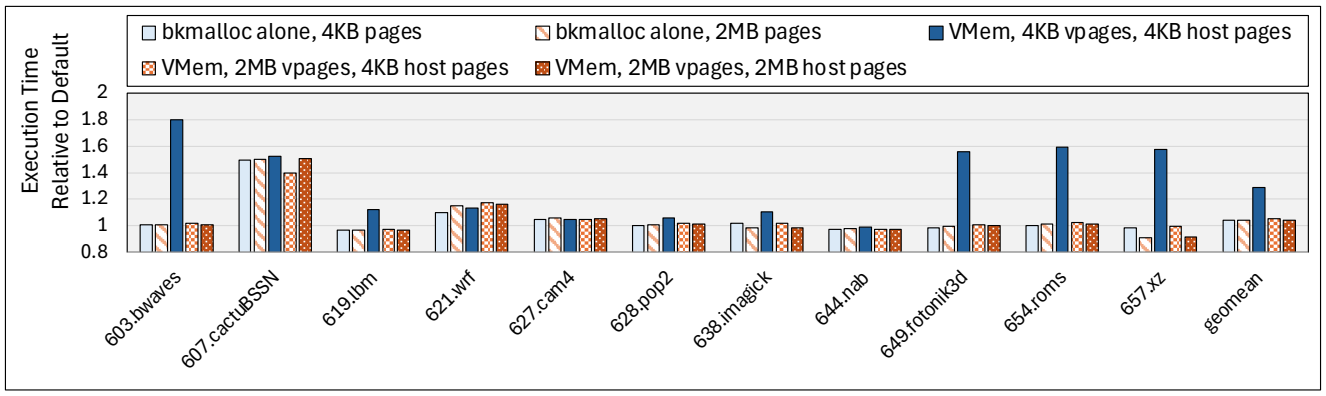
5.1 VMem Access Latency and Bandwidth

For our first set of tests, we employ a microbenchmark that maps a single anonymous region of size 1 GB and makes two linear passes over the region: one to populate (i.e., write) every word in the region with pseudorandom numeric data and a second that reads all of the words in the region and uses them to calculate a simple summation. Our experiments timed the performance of the write and read operations with the five configurations shown in Table 2. The results presented in the last four columns of Table 2 show the performance of each configuration in terms of both per-page latency (in μs) and throughput (in GB/s).

As expected, we find that VMem's reliance on `userfaultfd` can introduce significant overheads for operations that generate frequent page faults. However, the read operations, which only access addresses that have already been mapped to physical memory, perform very similarly in both the default and VMem configurations. Moreover, the additional fault overhead can be partially or almost entirely eliminated for applications that use larger page sizes. In these configurations, the time to write a full page of data far outweighs the additional costs imposed by `userfaultfd`, and thus, their impact is significantly diminished.



(a) Single-threaded FP benchmarks (*fprate*).



(b) Multi-threaded benchmarks (*fpspeed* + *657.xz_s*). Each benchmark uses 24 threads and runs on 24 cores.

Figure 2: Execution times of the selected SPEC® CPU 2017 benchmarks with bkmalloc and VMem configurations relative to the execution times of the default configuration with 4 KB pages (lower is better).

5.2 VMem Overheads with Real Applications

For many real applications, the faulting overhead produced by VMem can be mostly, or entirely, overlapped with other useful activity. To demonstrate this effect, we conducted a set of experiments using two groups of benchmarks from SPEC CPU 2017 [5]: 1) a group of single-threaded floating point (FP) applications (*fprate*) and 2) a group of OpenMP enabled applications configured to use one thread per core (*fpspeed* + *657.xz_s*). To avoid NUMA effects in the second group of tests, we spawn 24 OpenMP threads per benchmark and restrict all of these threads to run on only one subNUMA-cluster of our platform.

Our tests with VMem used the callback feature of the bkmalloc allocator to dynamically register each *anonymous* memory region mapped into the application's address space with our VMem Runtime. VMem then handled faults for these anonymous regions using the free list allocator described in Section 4. The experiments were configured so that VMem could satisfy all the anonymous faults for each benchmark from the local memory pool without needing to invoke page reclamation at any point.

Figure 2 shows the execution times of our selected benchmarks with the different VMem configurations from Table 2 relative to our default configuration, which uses the default system allocator (Ubuntu GLIBC v. 2.35) and default Linux memory manager with 4 KB pages on our platform. To isolate the effects of VMem from the bkmalloc allocator, we also include two *bkmalloc* configurations, which each use the bkmalloc allocator with different page sizes, but do not register any allocations with VMem.

We find that using VMem to manage anonymous memory regions has little or no performance impact on most of the single-threaded *fprate* benchamrks. In the worst cases with 4 KB pages (i.e., with *503.bwaves* and *507.cactuBSSN*), we see slowdowns of 15% and 12%, respectively, compared to the default software stack. In contrast, many of the multi-threaded benchmarks exhibit more substantial performance degradations when using VMem with 4 KB vpages. These workloads generate page faults much more frequently than the single-threaded benchmarks, and thus, cannot always overlap the additional costs imposed by VMem with useful execution. However, in most cases these performance degradations can be overcome by configuring VMem to use larger page sizes.

Of course, for some applications and platforms, larger page sizes are impractical because they can harm cache locality or increase internal fragmentation, and thus, raise capacity requirements. For example, in these experiments, *503.bwaves* and *507.cactuBSSN* exhibit significant slowdowns with larger page sizes due to much

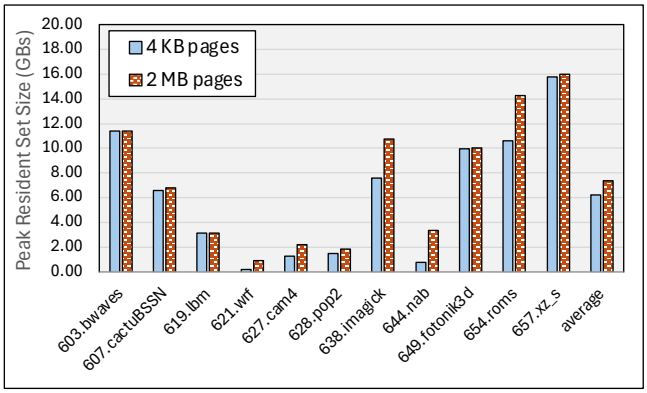


Figure 3: Peak resident set size (in GBs) of the multi-threaded benchmarks with the bkmalloc alone configuration with 4 KB and 2 MB page sizes.

worse L1 and L2 cache utilization. Additionally, we found that all of these benchmarks require some additional memory capacity with larger page sizes, but this effect is relatively muted (< 10%) in most cases (see Figure 3). On average, the memory capacity of these workloads increased by about 18% with larger pages. Whether these costs are feasible for a particular application or platform depends on a number of factors, including data layouts and the available memory capacity, but there are many real-world scenarios where larger page sizes are practical and beneficial. *Thus, these results are encouraging because they demonstrate that many applications and execution scenarios can leverage VMem without incurring prohibitive runtime costs.*

6 Leveraging VMem to Improve Memory Migration Throughput

Data migration costs are a significant challenge for complex memory platforms. Before moving any program data from one type of memory to another, the system must suspend any application threads that may access the data to prevent inconsistencies due to data races. Additionally, page tables must be updated and MMU caches (i.e., TLBs) flushed before the suspended activities can resume. On modern platforms, these costs are substantial and can hinder the performance of applications that must adapt to diverse or distributed memory hardware.

Some platforms, including the Intel Sapphire Rapids machine that we use in this work, include architectural features, such as vector instructions and hardware accelerators, that can increase copy throughput in memory. Additionally, and as discussed in Section 2, several recent projects have proposed system-level optimizations to reduce migration costs and enable applications to adapt more efficiently to complex memory hardware. One such optimization, which is part of the Carrefour algorithm [7], improves migration throughput by separating data copies from page table updates. In this scheme, the system optimistically write-protects and copies data to the target memory device asynchronously. If no writes to the original data are detected, the system can then complete the migration with only a short pause of the application threads to synchronize with page table updates. While these architectural features and system-level optimizations can effectively improve migration throughput for many workloads, most Linux platforms do not support them (even for huge page migrations) for a variety of reasons, including increased complexity and portability concerns.

To demonstrate the potential of VMem to reduce data migration costs, we extended the VMem Runtime and API with some additional features to enable *asynchronous* memory copy operations that are separate from page table updates, as described above. Additionally, our implementation of vregion migration in VMem uses the `memcpy` routine from GLIBC (v. 2.35) to copy data in memory from one vpage to another. On our platform, GLIBC was built and distributed with vector instructions enabled, and thus, VMem can potentially take advantage of these instructions during memory migration. Moreover, we extended VMem with an option (controlled via an environment variable) to offload vpage copying from the CPU and use our platform's DSA devices to accelerate vpage copying.

Table 3 presents the extended VMem API that we use for this study. [3] `vmem_mbind_vregion` is analogous to the `mbind` system call in Linux. It instructs the VMem runtime to fill demands in the given vregion with vpages corresponding to the specified device. `vmem_replicate_vregion_on_node` creates a copy of the vpages that have already been mapped into a given vregion using a new set of vpages on the given node. Note that this routine does not update the page table and that subsequent reads and writes to addresses within the vregion will resolve to the original vpages. In contrast, `vmem_migrate_vregion_to_node` completes the migration operation by remapping the page table entries corresponding to the given vregion and potentially creating a copy of its vpages on the given node, if necessary. Specifically, this routine invokes the `munmap` system call to unmap the host pages associated with the vregion and shoot down any cached page table entries in the processor TLBs. Next, if an up-to-date replica of the vregion already exists, then this routine will find and map the vregion addresses to the replicated vpages using the `mmap` system call. Otherwise, if this vregion was never replicated or if the original copy has been written since the time of replication, then it will create a copy of the necessary vpages on the given node prior to updating the page table. In these cases, its operation is thus similar to the Linux `mbind` with the `MPOL_MF_MOVE` flag set.

6.1 Evaluation of Enhanced Memory Migration with VMem

To demonstrate the potential of VMem to improve migration throughput, we designed a simple test program to migrate data between NUMA nodes on our Intel Linux platform. The test program creates and initializes two anonymous memory regions of size 4 GB: the first is initialized on the local memory node and the second is initialized on the remote memory node. For each region, the test program performs a simple sum over the data in the region, migrates the data in the region to the other memory node, and then recomputes the same sum after the data has been migrated. Additionally, it verifies that the region was actually migrated by

[3] As this work only intends to show and evaluate the potential of VMem to implement the data migration optimization described above, the current API only supports features that are necessary to demonstrate this approach. The API shown here will eventually be expanded to support additional use cases and production deployment of these features.

Function Name	Return Value	Arguments
vmem_mbind_vregion	int (success or error)	void *addr, size_t len, int node
vmem_replicate_vregion_on_node	int (success or error)	void *addr, size_t len, int node
vmem_migrate_vregion_to_node	int (success or error)	void *addr, size_t len, int node

Table 3: Extended VMem API for enabling data migration optimization.

measuring the time required to compute this sum, which differs based on whether the data accesses were resolved to the local or remote memory node.

We ran this experiment with three configurations:

(1) *Linux Migrate*: employs the system allocator to create memory regions and invokes the default system software stack (i.e., using the `mbind` system call) to migrate data between memory nodes.
(2) *VMem Migrate*: invokes `vmem_migrate_vregion_to_node` to create a copy of each vpage on the other memory device and update the page table as part of the same operation. For this configuration, we report the throughput of performing the data copy and page table update operations together.
(3) *VMem Optimized Migrate*: invokes `vmem_replicate_vregion_on_node` to replicate the entire vregion on the target memory and then separately invokes `vmem_migrate_vregion_to_node` to update the page table. For this configuration, we report the throughput of copying the data (*VMem Data Copy* as well as the throughput of updating the page table (*VMem Page Table Update*).

Additionally, we tested each configuration with both 4 KB and 2 MB page sizes as well as VMem configurations that use a 4 KB host page size with 2 MB vpages. For each VMem configuration, we also measured throughput with and without accelerating memory copies with the DSA.

First, with *VMem Optimized Migrate*, we find that VMem can effectively separates data copy and page table update operations. If an application continues to access virtual addresses that have been copied, but not remapped, these accesses will be resolved to physical memory corresponding to the original copy. Continued access to these addresses will resolve to the copied pages only after the application page tables have also been remapped.

Figure 4 presents the average throughput (in GB/s) of the data migration operations for each configuration. The results reveal several interesting observations. First, with the default Linux software stack, migration throughput improves with larger page sizes, but even with 2 MB page sizes, default Linux migrates less than 3 GBs of memory per second. With 4 KB host (Linux) pages and 4 KB vpages, the default VMem Migrate approach is about 13% slower than default Linux. This slowdown is attributable to the additional transfers between the host OS and VMem runtime that are necessary to complete migration operations for each vpage. However, with larger host and vpage sizes, VMem actually improves performance over default Linux because it can take advantage of vector instructions in the user-level `memcpy` implementation. Using the DSA within VMem can improve migration throughput even further and yields 1.6× and 9.4× speedups compared to default Linux with 4 KB and 2 MB pages, respectively.

Lastly, the measurements from the VMem Optimized Migrate configuration show that this approach has potential to improve migration throughput by multiple orders of magnitude in certain scenarios. While copy throughput is mostly similar to the default VMem Migrate configurations that perform data copy and page table remapping together, this optimization can overlap data copying with other useful application activity, in many cases. The operations that update the page tables, which do require synchronization with the application threads, are typically much faster than the standard migration operations. Indeed, if the replicated pages are already available on the target node, updating the page tables with VMem is over 1.5× faster with 4 KB pages and over 800× faster with 2 MB pages, compared to standard memory migration with default Linux. Thus, for many workloads and execution scenarios, VMem has significant potential to increase memory migration throughput and enable more efficient utilization of complex memory resources.

7 Discussion & Future Work

While this work has shown that the VMem framework has potential to enable powerful, portable optimizations for copying and moving data in a complex memory architecture, our work with VMem is still at an early stage. To realize the full potential of this approach, we plan to pursue several additional avenues of future research.

Our immediate future work is to build upon the replication-based optimization described above so that we can deploy it with real applications in both diverse and distributed memory architectures. While our current implementation does include features to track whether each vpage has been modified since the time it was replicated, this work only evaluates a scenario where the entire data set is unmodified after its been copied. Our next steps will investigate opportunities to apply this approach with real applications. Planned research tasks include: 1) identify scenarios where large data sets are likely to be unwritten or infrequently written for significant time intervals, 2) through automated compiler analyses or manual updates to source code, insert VMem API instructions to copy such data asynchronously at strategic locations within each application, 3) extend the VMem runtime with facilities to replicate vpages *automatically* when resources are available, and 4) investigate alternative memory management strategies that leverage data replication to free up capacity constrained memory resources quickly during periods of high demand (i.e., without needing to copy the data when demands rise).

Additionally, we plan to exploit deeper integration between applications and physical memory management to improve utilization of high performance memory resources with limited capacity. As described in Section 2, several prior works have found that leveraging application-level information (e.g., profiling and analysis of

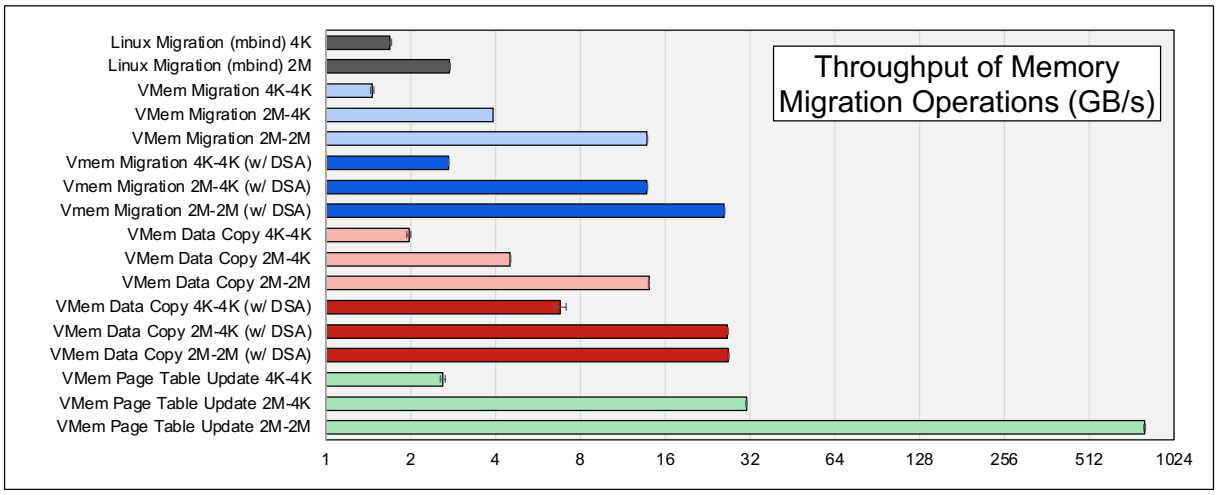


Figure 4: Throughput of memory migration operations with VMem (GB/s, log scale, higher is better). *Linux Migrate* shows memory migration throughput with the default Linux software stack. *VMem Migrate* shows the throughput of the data copy and page table update operations performed together in VMem. *VMem Data Copy* and *VMem Page Table Update* show the throughput of each operation separately, as done with the *VMem Optimized Migrate* configuration.

logical structures or objects) to help guide the process of matching application data to heterogeneous memory resources can improve performance significantly. By providing applications with fine-grained controls to define custom data tiering and management policies, VMem has potential to facilitate the discovery of new heuristics and algorithms to improve these processes.

Moreover, while our current VMem implementation maintains vpages corresponding to distinct NUMA nodes in separate memory pools, there are no fundamental restrictions on how the vpages in VMem are organized internally. In the future, we will extend VMem with new strategies that automatically sort vpages into distinct pools according to other microarchitectural features that are present in the underlying architecture, including cache sets and DRAM features, such as channels, ranks, banks, and row buffers.

There are numerous examples in the literature of techniques and optimizations that rely on fine-grained management of these sorts of features. Some specific examples include: application-guided page coloring to improve cache utilization [11, 27], co-locating data with similar reuse increase energy efficiency [14, 24], and managing allocations across DRAM banks to improve row buffer utilization [30] or provide bandwidth guarantees [34]. Such controls are also essential for emerging processing-in-memory (PIM) systems, where data must be structured properly within the device to take advantage of in-memory computing [12, 20]. While previous efforts have demonstrated the benefits of these techniques, they are still not widely used because their implementations rely on system features or modifications that are not available on most platforms. By empowering applications with fine-grained and cross-platform controls, VMem can enable more portable implementations of these existing techniques and also facilitate the discovery of new optimizations to improve utilization of modern memory hardware.

8 Conclusion

This work presents VMem: a novel runtime framework for enabling application control of physical memory resources. VMem allows applications to exert direct control over system-level memory management tasks, including translation of virtual addresses, physical memory allocation and recycling, and memory-to-memory migration. Its Linux-based implementation provides these features without requiring any kernel modifications, custom hardware, or non-standard system configurations. Using microbenchmarks as well as real applications from SPEC CPU, the evaluation characterizes the performance of VMem operations and demonstrates its potential to reduce pause times for applications that migrate data frequently on complex architectures. Overall, this work shows there is great opportunity in using VMem to unlock new optimizations and efficiencies by reducing the semantic gap between system and application software during memory management.

References

[1] Agarwal, N., and Wenisch, T. F. 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* (New York, NY, USA, 2017), ASPLOS '17, ACM, pp. 631–644.

[2] Akram, S., Sartor, J. B., McKinley, K. S., and Eeckhout, L. Write-rationing garbage collection for hybrid memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2018), PLDI 2018, Association for Computing Machinery, p. 62–77.

[3] Anonymous. bkmalloc memory allocator. URL will be provided when submission is unblinded, January 2025.

[4] Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. Dune: safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (USA, 2012), OSDI'12, USENIX Association, p. 335–348.

[5] Bucek, J., Lange, K.-D., and v. Kistowski, J. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (New York, NY, USA, 2018), ICPE '18, Association for Computing Machinery, p. 41–42.

[6] Choi, J., Blagodurov, S., and Tseng, H.-W. Dancing in the dark: Profiling for tiered memory. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (Portland, OR, USA, 2021), IEEE, pp. 13–22.

[7] Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., and Roth, M. Traffic management: a holistic approach to memory placement on numa systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, Association for Computing Machinery, p. 381–394.

[8] Dulloor, S. R., Roy, A., Zhao, Z., Sundaram, N., Satish, N., Sankaran, R., Jackson, J., and Schwan, K. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, Association for Computing Machinery.

[9] Elphinstone, K., and Heiser, G. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, Association for Computing Machinery, p. 133–150.

[10] Engler, D. R., Kaashoek, M. F., and O'Toole, 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, Association for Computing Machinery, p. 251–266.

[11] Guo, R., Liao, X., Jin, H., Yue, J., and Tan, G. Nightwatch: integrating lightweight and transparent cache pollution control into dynamic memory allocation systems. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (USA, 2015), USENIX ATC '15, USENIX Association, p. 307–318.

[12] Ibrahim, M. A., Islam, M., and Aga, S. PIMnast: Balanced Data Placement for GEMV Acceleration with Processing-In-Memory . In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, Nov. 2024), IEEE Computer Society, pp. 970–981.

[13] Jalalian, S., Patel, S., Hajidehi, M. R., Seltzer, M., and Fedorova, A. Extmem: enabling application-aware virtual memory management for data-intensive applications. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference* (USA, 2025), USENIX ATC'24, USENIX Association.

[14] Jantz, M. R., Robinson, F. J., Kulkarni, P. A., and Doshi, K. A. Cross-layer memory management for managed language applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2015), OOPSLA 2015, ACM, pp. 488–504.

[15] Kammerdiener, B., McMichael, J. Z., Jantz, M., Doshi, K., and Jones, T. Flexible and effective object tiering for heterogeneous memory systems. *ACM Trans. Archit. Code Optim.* (Dec. 2024). Just Accepted.

[16] Kim, J., Choe, W., and Ahn, J. Exploring the design space of page management for Multi-Tiered memory systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (virtual, July 2021), USENIX Association, pp. 715–728.

[17] Kleen, A. A numa api for linux, August 2004.

[18] Kravetz, M. The linux kernel documentation: Hugetlbfs reservation. https://www.kernel.org/doc/html/v4.20/vm/hugetlbfs_reserv.html, April 2017.

[19] Laghari, M., Ahmad, N., and Unat, D. Phase-based data placement scheme for heterogeneous memory systems. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* (Lyon, France, 2018), IEEE, pp. 189–196.

[20] Lee, S., Kang, S.-h., Lee, J., Kim, H., Lee, E., Seo, S., Yoon, H., Lee, S., Lim, K., Shin, H., Kim, J., O, S., Iyer, A., Wang, D., Sohn, K., and Kim, N. S. Hardware architecture and software stack for pim based on commercial dram technology. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (2021), ISCA '21, IEEE Press, p. 43–56.

[21] Liedtke, J. Improving ipc by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1993), SOSP '93, Association for Computing Machinery, p. 175–188.

[22] Liedtke, J. Toward real microkernels. *Commun. ACM 39*, 9 (Sept. 1996), 70–77.

[23] Olson, M. B., Kammerdiener, B., Jantz, M. R., Doshi, K. A., and Jones, T. Online application guidance for heterogeneous memory systems. *ACM Trans. Archit. Code Optim. 19*, 3 (jul 2022).

[24] Olson, M. B., Teague, J. T., Rao, D., JANTZ, M. R., Doshi, K. A., and Kulkarni, P. A. Cross-layer memory management to improve dram energy efficiency. *ACM Trans. Archit. Code Optim. 15*, 2 (May 2018), 20:1–20:27.

[25] Peng, I., McFadden, M., Green, E., Iwabuchi, K., Wu, K., Li, D., Pearce, R., and Gokhale, M. Umap: Enabling application-driven optimizations for page management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)* (Nov 2019), pp. 71–78.

[26] Peng, I. B., Gioiosa, R., Kestor, G., Cicotti, P., Laure, E., and Markidis, S. Rthms: A tool for data placement on hybrid memory system. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management* (New York, NY, USA, 2017), ISMM 2017, ACM, pp. 82–91.

[27] Qureshi, M. K., and Patt, Y. N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)* (2006), pp. 423–432.

[28] Raybuck, A., Stamler, T., Zhang, W., Erez, M., and Peter, S. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 392–407.

[29] Servat, H., Peña, A. J., Llort, G., Mercadal, E., Hoppe, H., and Labarta, J. Automating the application data placement in hybrid memory systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)* (Hawaii, USA, Sept 2017), IEEE, pp. 126–136.

[30] Sudan, K., Chatterjee, N., Nellans, D., Awasthi, M., Balasubramonian, R., and Davis, A. Micro-pages: increasing dram efficiency with locality-aware data placement. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2010), ASPLOS XV, Association for Computing Machinery, p. 219–230.

[31] Verma, V. Intel tiering patches for linux, February 2022.

[32] Wu, K., Huang, Y., and Li, D. Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2017), SC '17, ACM, pp. 58:1–58:14.

[33] Yan, Z., Lustig, D., Nellans, D., and Bhattacharjee, A. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2019), ASPLOS '19, Association for Computing Machinery, p. 331–345.

[34] Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., and Sha, L. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2013), pp. 55–64.