FBMM: Using the VFS for Extensibility in Kernel Memory Management

Bijan Tabatabai
bijan@cs.wisc.edu
University of Wisconsin - Madison

Mark Mansi
markm@cs.wisc.edu
University of Wisconsin - Madison

Michael M. Swift
swift@cs.wisc.edu
University of Wisconsin - Madison

ABSTRACT

Modern memory hierarchies are increasingly complex, with more memory types and richer topologies. Unfortunately kernel memory managers lack the extensibility that many other parts of the kernel use to support diversity. This makes it difficult to add and deploy support for new memory configurations, such as tiered memory: engineers must navigate and modify the monolithic memory management code to add support, and custom kernels are needed to deploy such support until it is upstreamed.

We take inspiration from filesystems and note that VFS, the extensible interface for filesystems, supports a huge variety of filesystems for different media and different use cases, and importantly, has *interfaces for memory management operations* such as controlling virtual-to-physical mapping and handling page faults.

We propose writing memory management systems as filesystems using VFS, bringing extensibility to kernel memory management. We call this idea File-Based Memory Management (FBMM). Using this approach, many recent memory management extensions, e.g., tiering support, can be written without modifying existing memory management code. We prototype FBMM in Linux to show that the overhead of extensibility is low (within 1.6%) and that it enables useful extensions.

ACM Reference Format:
Bijan Tabatabai, Mark Mansi, and Michael M. Swift. 2023. FBMM: Using the VFS for Extensibility in Kernel Memory Management . In *Workshop on Hot Topics in Operating Systems (HOTOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3593856.3595908

HOTOS '23, June 22–24, 2023, Providence, RI, USA

ACM ISBN 979-8-4007-0195-5/23/06…$15.00
https://doi.org/10.1145/3593856.3595908

1 INTRODUCTION

In modern operating systems, the model of physical memory is caches backed by main memory, which may be in multiple NUMA nodes, which itself may be backed by a swap device. Linux, for example, provides a generic NUMA *distance* metric [21]. However, with the introduction of CXL [22], richer physical memory configurations such as tiered or disaggregated memory and multiple memory types are gaining popularity, but do not fit easily into that model. While operating systems are being modified to support these memory configurations [12, 17], the changes are cumbersome because memory management code is spread across many different parts of the kernel and provides no extensibility support. For example, transparent huge page support is spread across 18 source files.

This problem has two primary sources. First, Linux has a limited abstraction of physical memory: a set of pages, possibly split into NUMA nodes and/or backed by swap space, which does not capture the rich variety of memory types or the relationships between those memory types, such as asymmetric read/write performance. Second, the Linux kernel memory management code is monolithic and provides no extensibility to adapt mechanisms or policies for new memory types and management practices; past work directly modifies core Linux memory management code [10].

We observe that the VFS layer addresses a similar problem for filesystems: all filesystems are accessed using generic system calls, which then call the corresponding filesystem code that can use workload- or device-specific policies and mechanisms. This extensibility has lead to a flowering of filesystems supporting widely varying devices and usages, such as filesystems for non-volatile memory (e.g., PMFS), flash storage (F2FS) or DRAM (RAMFS). Notably the VFS layer provides interfaces for memory-mapped files that control the mapping of virtual addresses to data.

Many previous systems explored making memory management extensible [3, 11, 18, 23]. However, the concepts for extensibility used in these systems, focus on application-specific paging policies rather than new memory hardware, and generally have not been adopted by Linux or other operating systems.

To address this need, we propose that the kernel memory manager should be made extensible to allow new ways of managing physical memory, virtual addresses, and page movement; and that in Linux, the VFS filesystem interface can be used for memory management extensions. Extensibility should provide clean interfaces so that developers can write new memory managers without modifying existing code. In addition, it should allows memory managers to be written as a separate module, allowing them to be deployed without needing a custom kernel.

While the OS community is fond of designing new interfaces, we propose to instead *use VFS as an extension interface* because it *already exists* and is *good enough* for many uses. It may not provide every possible form of extension, but it has sufficient power to support many recent memory management modifications.

We call this design File-Based Memory Management (FBMM). With FBMM, memory managers are written as filesystems. Allocating/freeing memory corresponds to creating/extending and deleting/truncating a file. We term these memory-management file systems (MMFSs). An MMFS can control how its gets physical memory (e.g., reserved at boot, requested from the kernel at some granularity), how and when it is allocated (on map, on fault), and how allocations are placed in physical memory.

FBMM supports two usage modes. To support unmodified applications, FBMM allows specifying the default memory manager for a process, to which all requests for anonymous memory are passed. For finer-grained control or to allow different memory managers for different data structures, applications can also explicitly call an MMFS to request memory for the structure. Precedent for this paradigm already exists in Linux in HugeTLBFS, where memory backed by huge pages is allocated by creating and mapping files in the filesystem, and had a similar motivation: how to support huge pages without complicating the existing memory management code [13].

In this work, we motivate extensible memory management by showing that the current approach is unsustainable (Section 2). We then describe the design of FBMM, and demonstrate how it can be used to support proposed memory management extensions currently implemented as modifications to Linux kernel memory management (Section 3). The overhead of this abstraction is small (Section 4): in our proof of concept based on `ext4 DAX`, the throughput of Memcached is within 1.6% compared to using normal Linux's memory management. We also use FBMM to build from scratch a functional (but limited) tiered memory management system in under 1500 lines of code.

2 MOTIVATION AND RELATED WORK

Unlike much of the kernel, memory management code is still largely monolithic with few interfaces for extensibility. As a result, support for new memory management policies and mechanisms generally require wide-ranging and invasive code changes. Making such changes is difficult: it requires detailed knowledge of the existing memory management subsystem to identify *all* relevant places to change and knowledge of the complex data structures and locking protocols.

For example, support for transparent huge pages in Linux is distributed across 18 files, and impacts wide ranging concerns including page-fault handling, physical memory allocation, page table management, etc. Furthermore, the policies controlling when to use huge pages are widely distributed, increasing the likelihood of pathological behavior and long-latency operations [14].

New Memory Architectures Compute Express Link (CXL) promises to simplify hardware support for new memory architectures, such as heterogeneous or tiered memory. However, the non-extensible nature of the memory management code complicates adding software support for these configurations. The existing abstraction of NUMA nodes can be used by representing the slow memory tier as a CPU-less node. However, prior work has shown this is suboptimal [17] because the policy for NUMA migration is largely based on attracting hot memory to the local node, while tiering often migrates cold memory to slower memory devices. Meta's Transparent Page Placement (TPP) adds support for tiered memory in Linux by modifying the NUMA system to better fit its needs [17], but at the cost of modifying 22 files [16]. Similarly, support for hardware with configurable channel mappings [24] also required kernel changes to provide page allocators for different configurations of memory. An extensible interface for memory management could have simplified each of these designs by keeping the implementation separate from the rest of the memory management code.

Extensible Memory Management Making the memory system more extensible is not a new idea. Mach [18], VINO [23], Nemesis [9], and Krueger et al. [11] introduce systems that allow applications to modify their paging behavior. SPIN [3] allows applications to register callback functions on memory management events for extensibility. However, both paging and callbacks are not rich enough to describe many memory configurations, like tiered memory, as they cannot be used for asynchronous operations like hotness tracking and page migration. HeMem [19] extends the memory system in a user library that implements tiered memory, but this approach lacks the control and information that would be allowed in the kernel, such as access to LRU lists.

Linux does provide limited extensibility in the memory reclamation system. The `shrinker` interface allows kernel modules with caches to be notified that they should release memory [5]. The `frontswap` interface allows pages to be swapped out to alternate media, such as compressed or remote memory [1, 8]. Neither interface allows customization of when or which pages are reclaimed, or where pages are allocated.

Prior Successes The designers of NFS solved a similar problem of extensibility in filesystems to allow for transparent access to remote files [20]. Their solution was to create the Virtual Filesystem (VFS) layer that abstracts generic filesystem system calls, like `open` or `write`, from the implementation of those operations that are specific to a filesystem itself. The addition of the VFS layer makes it easier to create new filesystems by implementing a set of callback functions rather than modifying more general filesystem code. Today, the Linux kernel has around 50 filesystem implementations in-tree.

Beyond filesystems, VFS has been used to implement memory managers. Prior to support for transparent huge pages, HugeTLBFS provided a filesystem based memory allocator that enabled applications to explicitly request huge-page backed memory. HugeTLBFS relies on the VFS as an extension mechanism, and requires only minor changes to existing memory management code. This noninvasive approach was essential for allowing HugeTLBFS into the kernel which enabled Linux to support huge pages several years before transparent huge pages were supported [13]. The success of HugeTLBFS as a deployable huge-page mechanism points to a potential solution to extensibility: use the power of the VFS as an extension mechanism to support richer memory management mechanisms and policies and more varied memory configurations. However, it also provides a lesson: extension systems must be transparently usable by applications to provide real value, and HugeTLBFS had to be replaced because it required application modifications (among other reasons).

To more easily extend the memory management system, operating systems need an abstraction between the generic memory management operations provide by the memory manager, and the hardware- and policy-specific implementations of those operations.

3 DESIGN

We propose an extensible memory management architecture for Linux. Our proposal has the following goals:

(1) New memory management systems modifying physical memory management, virtual address management, or page movement can be written as standalone pieces of software, i.e., without modifying the kernel (*extensibility*).

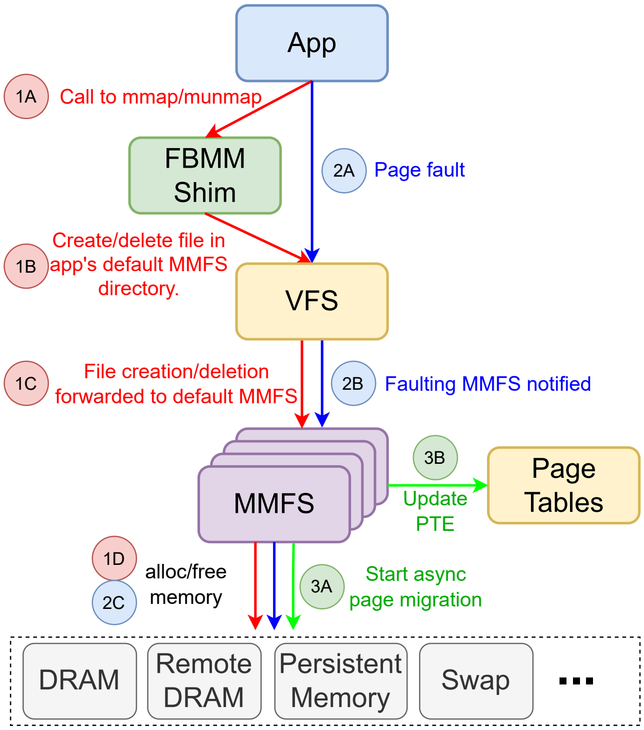

Figure 1: The overall architecture of FBMM.

(2) Applications can use a default memory manager without code changes (*transparency*).

(3) Sophisticated applications can use multiple memory managers on different data structures at the same time (*control*).

We propose writing memory management systems as filesystems: rather than allocating memory using a generic kernel allocator, processes instead create and map files through VFS that provide access to memory. An application chooses which memory manager to use by choosing where it creates files. As a result, the memory for a process is made of files that have been memory mapped into the process. We call this paradigm File-Based Memory Management (FBMM).

Design Overview Figure 1 shows our design. The system comprises the FBMM Shim (described below) and one or more Memory-Management File Systems (*MMFSs*) that provide different memory management mechanisms and policies. Every process is assigned a MMFS to act as its default memory manager (e.g., through an environment variable), but can explicitly use any loaded MMFS.

In FBMM, applications map anonymous virtual memory, via `mmap` with the `MAP_ANON` flag or `brk`, or unmap anonymous memory, via `munmap`, the same way they do now (1A). However, the FBMM Shim intercepts these calls and either creates and then maps a file in the application's default MMFS when mapping memory, or deletes the files associated with a memory range when unmapping memory (1B). We call such files *memory files*. The creation/deletion of files invokes the VFS layer, which forwards the operations to the relevant MMFS (1C). Finally, the MMFS updates its own metadata/bookkeeping as needed and allocates (if using eager paging) or frees the physical memory for the created/deleted file (1D).

The FBMM Shim provides transparency to applications by routing memory management system calls to file operations in the application's default MMFS. However, an advanced application may want a specific memory region to have different behavior than the default. For example, an application might want a DMA buffer to be pinned in memory, while the rest of its memory can be movable. An application can accomplish this by manually creating a memory file in the MMFS that has its desired behavior for a region and mapping it to its address space. Note that when a memory file is created manually, it is not tracked by the FBMM Shim. Therefore, its memory is not freed automatically on a call to munmap. Instead, the memory is freed when the file is closed, either manually with the close syscall or by the kernel when the application terminates.

On a page fault, the page fault handler directly invokes VFS, using information about the faulting memory file (2A). Like before, the VFS then notifies the MMFS owning the faulty-containing file (2B). Then, the MMFS handles the page fault in the manner it sees fit, such as allocating physical memory for the fault (2C).

An MMFS can also execute asynchronously to MM events. For example, a tiered memory MMFS can initiate page migration asynchronously without any outside prompting (3A). Furthermore, an MMFS can utilize any useful function that the existing Linux memory management code exposes. In our tiered memory example, the MMFS uses relevant page table walking and updating functions to update the physical address of the migrated page (3B).

Interfaces A strict subset of callback functions defined by the VFS layer suffices to define an MMFS. The major callbacks defining an MMFS are listed in Table 1. With these interfaces, a MMFS can control when memory is allocated (via mmap), where memory is allocated in the virtual address space (get_unmapped_area), what physical memory backs an allocation (fallocate and fault). This is not an exhaustive list of the callbacks an MMFS can implement, but are highlighted to show the richness of the interface.

The functionality of a MMFS is not limited to the callbacks provided by the VFS interface. Filesystems may do work asynchronously via a kernel thread. For example, a MMFS may perform page migration without prompting from the VFS or a sysfs interface.

Additional, MMFSs can call existing Linux memory management code. For example, to allocate physical memory, an MMFS can statically provision memory at boot time, or can call get_free_pages to allocate physical memory dynamically. An MMFS can also search a process's vm_area_struct trees, or modify its page tables directly. Generally, an MMFS can use any helper functions and data structures exported by existing memory management code.

Memory operations A MMFS is available for use when it is mounted. All files created under its mount point allocate memory using the MMFS. When a process starts, the parent process or system administrator can specify a directory indicating the MMFS to use for anonymous memory management. If no directory is specified, a system default is used.

When a process calls brk or mmap with the MAP_ANON flag, the FBMM Shim creates an unnamed temporary file in the mount directory of the process's assigned MMFS and maps the file to the creating process' address space. The FBMM Shim also saves a reference to the file in a per-process tree indexed by the virtual address range it maps. When a process terminates or calls munmap, the FBMM Shim searches the process's tree of mapped files and deletes the files in the region being unmapped. This triggers the MMFS deletion procedure, which frees the physical memory. Existing Linux functionality handles faults to memory files: the kernel identifies and invokes the MMFS corresponding to a memory region the first time a page is touched.

Discussion FBMM meets our design goals in the following ways:

(1) *Extensibility:* Like normal filesystems, a MMFS can be written as a kernel module that does not require kernel modifications. Through choice of when and which pages to allocate, it controls physical memory. Through mapping operations, it controls virtual addresses. Through asynchronous execution, it can perform background page movement.
(2) *Transparency:* The default memory manager for an application can be changed by pointing the process to the mount directory of a different MMFS without changing the application.
(3) *Control:* Applications can manually create and manage files in a different MMFS.

Design examples Our design is expressive enough to be used to reimplement existing systems as MMFSs.
Tiering. For example, to implement TPP [17] as an MMFS one needs to be able to choose whether a new page should be allocated between near and far memory, measure which pages are hot and cold, and migrate pages from near to far memory and vice versa. The decision of where to place a new page happens in callbacks responsible for allocating

Table 1: Interfaces used by MMFSs.

Interface	Defined in	Called by	Purpose
mmap (callback, not syscall)	struct file_operations	Application via VFS	provide VFS a set of functions (struct vm_operations_struct) to manage a mapping
get_unmapped_area	struct file_operations	mmap syscall	allocate virtual address range
fault	struct vm_operations_struct	Page fault handler	control the paging behavior of a process
fallocate	struct file_operations	mmap syscall with MAP_POPULATE flag	signal need to allocate physical memory
free_inode	struct super_operations	file deletion code	signal need to free physical memory

physical memory, i.e. fault and fallocate. Hotness tracking and migration are asynchronous tasks that do not correspond to a callback and instead execute in a kernel thread. Hotness tracking periodically scanning page table access bits. Migration checks the page hotness information to determine promotion/demotion candidates, copying memory to its new home, and then updating the page table to reflect the changes.

Address translation. Another example is ASAP [15] and Redundant Memory Mappings [10], which propose changes to how page tables are allocated and managed. When written as an MMFS, the code to allocate the page tables specific to the memory system would be placed inside the fault callback.

New policies. CBMM [14] proposes using cost-benefit models to make memory management decisions, such as whether or not to allocate a huge page, eagerly allocate a page before its first access, or prezero some number of pages. In the original implementation, calls to the cost-benefit models were placed throughout the memory management code; however, the implementation could instead be localized inside of an MMFS. Huge page decisions are made inside of the fault and fallocate callbacks. Eager paging is done inside the mmap callback, where the model decides which pages of a mapping should be preallocated. This is in contrast to mapping anonymous memory with the MAP_POPULATE flag which preallocates every page. Prezeroing, an asynchronous operation, is done by a kernel thread.

Limitations. One limitation of our design is that there is no way for the MMFSs in the system to coordinate with each other. Such coordination would be useful for dealing with global memory pressure, for example. An MMFS can register a "shrinker" to free its caches and swap out pages when the kernel requires more memory; however, the kernel invokes these shrinkers in an arbitrary order. It may be useful to have a mechanism where the MMFSs could coordinate to decide the ideal order to shrink them.

Another limitation is that our design does not consider extensions to the memory management API exposed to applications, such as what is described in mmapx [2].

4 PROOF OF CONCEPT

We implemented the FBMM Shim in Linux kernel version 5.14 and created a simple proof of concept MMFS based on ext4 Direct Access (DAX). We compare the cost of memory operations against the standard kernel memory manager.

Implementation. We implemented the FBMM Shim as additional functionality within the Linux memory manager. It exposes interfaces to the Linux memory manager to construct its per-process tree of memory files and redirect anonymous memory operations to the appropriate MMFS. The FBMM Shim has four main functions it exposes to the rest of Linux:

- fbmm_create_new_file: Creates a new, unnamed temporary file and returns a handle to that file to the caller.
- fbmm_register_new_file: Adds a newly mapped file to the per-process tree of memory files, indexed by the virtual address it is mapped to.
- fbmm_munmap: Invoked when a process calls munmap to delete or truncate the memory file.
- fbmm_exiting_proc: Called when a process exits and deletes any remaining files attributed to that process.

We made minor modifications the existing Linux memory manager to support FBMM. When the mmap syscall is called with the MAP_ANON flag, it creates a file via fbmm_create_new_file and then maps that file. Then, after the new file was mapped, it calls fbmm_register_new_file was made to track the new file. The brk system call was similarly modified. The munmap and exit system calls invoke fbmm_munmap and fbmm_exiting_proc respectively to do cleanup.

Prototype MMFS. We built a proof of concept MMFS on top of ext4 DAX, a variant of ext4 intended for persistent memory and byte-addressable direct access to the storage medium [6]. Instead of persistent memory, we back the ext4 DAX filesystem with DRAM reserved at boot time. We use this rather than RAMFS, because ext4 implements its own allocator rather than relying on the kernel page allocator. In addition, as a DAX filesystem it bypasses the page cache to allow applications to map file data directly to physical location in memory. These implementation shortcuts allow us to focus on the design and implementation of the FBMM Shim and use the ext4 DAX filesystem without modification to test and evaluate FBMM. We disable metadata and journaling in ext4 DAX, as we do not need persistence. Because ext4 DAX was designed for file storage, rather than memory management, its performance gives an upper bounds of the overhead of memory management via FBMM – an optimized MMFS would likely have lower overhead. We observed a few cases where ext4 was faster than the kernel memory manager, and made a handful of small optimizations to the Linux memory management code to remove these anomalies.

Preliminary Results. We ran two sets of experiments to measure the overhead of the FBMM Shim and VFS layer. First, we compare how quickly Linux's memory management system and FBMM with ext4 allocate and map memory. Second, we measure the performance of Memcached using the ext4 MMFS using YCSB [4] and compare against Linux on the same workload. Our experiments were run on Linux 5.14 on bare metal Cloudlab [7] xl170 machines with 64GB of ECC DDR4, and ten 2.4GHz Intel Broadwell cores. Experiments were run both with and without huge pages enabled.

The allocation benchmark allocates and populates a 32GB memory region using the MAP_POPULATE flag. With huge pages, the FBMM system is 2% slower than standard Linux due to bookkeeping done by the DAX subsystem; the FBMM Shim adds no overhead. With base pages, FBMM is 50% slower than Linux due to the overhead in ext4 DAX of allocating single pages – an artifact of its design for file workloads not memory management. A more optimized MMFS would remove this overhead. For huge pages, the cost of allocating a block is overshadowed by the time to zero memory.

For our second set of experiments, we looked at the throughput of Memcached operations for read-only, read/write, and insert-only workloads driven by YCSB, where all heap memory is allocated by the ext4 DAX MMFS. In all three workloads, the FBMM system is within 1% of the performance of the standard Linux system when huge pages are used. Surprisingly, when base pages are used, the FBMM system is within 1.6% of the performance of Linux, indicating that the workload is not very sensitive to page allocation cost.

Overall, despite the fact that our proof of concept is not at all tuned for memory management, it does not significantly degrade the end-to-end performance of Memcached.

Tiering Extension. To evaluate the extensibility and richness of our design, we are in the process of building a tiered memory manager similar to TPP [17]: it seeks to place cold data in a slower (more distant) tier of memory while keeping hot memory close to the processor and is suitable for CXL-based disaggregated memory systems.

Our implementation keeps a list of hot and cold pages of allocated memory for both the fast and slow tiers of memory. If there is adequate space in the fast tier when memory is requested, the MMFS services the allocation with the fast tier; otherwise, the allocation is serviced by the slow tier. The MMFS adds the corresponding page to the hot list of the page's tier. Periodically, a kernel thread scans through the hot and cold lists checking the page table access bits of the pages. The same kernel thread monitors the amount of free memory in the fast tier, and migrates pages to maintain enough free memory.

Though not fully evaluated, our tiering MMFS is functional and is able to dynamically detect and migrate hot/cold pages to the appropriate fast/slow memory. The system was created as a standalone kernel module, and is only about 1500 lines of C code in total, much of which is debugging and boilerplate code for defining a filesystem.

5 CONCLUSION

Memory management extensibility has been a long standing issue in operating systems research [3, 11, 18, 23]. However, the memory management system in the Linux kernel provides few interfaces for extensibility. The emergence of new memory technology and topologies like tiered and disaggregated memory has put new pressure on this front. To solve this problem, we propose using the extensibility provided by VFS to create extensible memory management software. Our prototype implementation demonstrates that the overhead of invoking a file system can be low, and that the VFS interface has the expressiveness for important extensions like tiering.

ACKNOWLEDGEMENTS

This work was supported in part by PRISM, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by NSF grants CNS 1815656 and CNS 1900758.

REFERENCES

[1] Linux Kernel Documentation: Zswap. https://www.kernel.org/doc/Documentation/vm/zswap.txt.

[2] Reto Achermann, David Cock, Roni Haecki, Nora Hossle, Lukas Humbel, Timothy Roscoe, and Daniel Schwyn. mmapx: Uniform memory protection in a heterogeneous world. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, 2021.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP '95, 1995.

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

[5] Jonathan Corbet. Smarter Shrinkers. https://lwn.net/Articles/550463/, May 2013.

[6] Tom Coughlan. Persistent Memory in Linux. https://www.snia.org/sites/default/files/PM-Summit/2017/presentations/Coughlan_Tom_PM_in_Linux.pdf, 2017. SNIA Peristent Memory Summit.

[7] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference*, USENIX ATC '19, July 2019.

[8] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, 2017.

[9] Steven M Hand. Self-Paging in the Nemesis Operating System. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, OSDI '99, 1999.

[10] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, June 2015.

[11] Keith Krueger, David Loftesness, Amin Vahdat, and Thomas Anderson. Tools for the Development of Application-Specific Virtual Memory Management. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, 1993.

[12] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, 2023.

[13] Adam G Litke. "Turning the Page" on Hugetlb Interfaces. In *Proceedings of the Linux Symposium*, page 277, 2007.

[14] Mark Mansi, Bijan Tabatabai, and Michael M Swift. CBMM: Financial Advice for Kernel Memory Managers. In *2022 USENIX Annual Technical Conference*, USENIX ATC '22, 2022.

[15] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, 2019.

[16] Hasan Al Maruf. [PATCH 0/5] Transparent Page Placement for Tiered-Memory. https://lore.kernel.org/lkml/cover.1637778851.git.hasanalmaruf@fb.com/.

[17] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, 2023.

[18] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, 1987.

[19] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, 2021.

[20] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the summer 1985 USENIX conference*, 1985.

[21] Steve Scargall. Linux NUMA Distances Explained. https://stevescargall.com/2022/11/03/linux-numa-distances-explained/, November 2022.

[22] Debendra Das Sharma. Compute Express Link®: An open industry-standard interconnect enabling heterogeneous data-centric computing. In *2022 IEEE Symposium on High-Performance Interconnects*, HOTI, 2022.

[23] Christopher A Small and Margo I Seltzer. Vino: An Integrated Platform for Operating System and Database Research. Technical Report TR-30-94, Harvard University, 1994.

[24] Jialiang Zhang, Michael M. Swift, and Jing Jane Li. Software-Defined Address Mapping: A Case on 3D Memory. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, 2022.