FBMM: Making Memory Management Extensible With Filesystems

Bijan Tabatabai, James Sorenson and Michael M. Swift,
University of Wisconsin–Madison

https://www.usenix.org/conference/atc24/presentation/tabatabai

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

FBMM: Making Memory Management Extensible With Filesystems

Bijan Tabatabai
bijan@cs.wisc.edu

James Sorenson
jaso@cs.wisc.edu

Michael M. Swift
swift@cs.wisc.edu

University of Wisconsin-Madison

Abstract

New memory technologies like CXL promise diverse memory configurations such as tiered memory, far memory, and processing in memory. Operating systems must be modified to support these new hardware configurations for applications to make use of them. While many parts of operating systems are extensible, memory management remains monolithic in most systems, making it cumbersome to add support for a diverse set of new memory policies and mechanisms.

Rather than creating a whole new extensible interface for memory managers, we propose to instead use the memory management callbacks provided by the Linux virtual file system (VFS) to write memory managers, called memory management filesystems (MFSs). Memory is allocated by creating and mapping a file in an MFS's mount directory and freed by deleting the file. Use of an MFS is transparent to applications. We call this system *File Based Memory Management* (FBMM).

Using FBMM, we created a diverse set of standalone memory managers for tiered memory, contiguous allocations, and memory bandwidth allocation, each comprising 500-1500 lines of code. Unlike current approaches that require custom kernels, with FBMM, an MFS can be compiled separately from the kernel and loaded dynamically when needed. We measure the overhead of using filesystems for memory management and found the overhead to be less than 8% when allocating a single page, and less than 0.1% when allocating as little as 128 pages. MFSs perform competitively with kernel implementations, and sometimes better due to simpler implementations.

1 Introduction

For decades, the memory hierarchy of computer systems was fixed as processor caches, backed by byte addressable and volatile main memory, which itself may be backed by block level nonvolatile storage. However, new hardware technologies, such as huge local memories, nonvolatile byte addressable memories, and CXL-attached far memory, have emerged in recent years that change the traditional memory hierarchy. These hardware technologies have inspired a myriad of systems research for memory management (MM) extensions like contiguous allocation [5], tiered memory [20, 33, 38] and disaggregated memory [26, 41]. Linux developers have implemented tightly integrated support for the most available technologies, such as NUMA zones and transparent huge pages as part of the monolithic kernel MM subsystem.

As the number of new hardware mechanisms grows, and the set of desirable mechanisms and policies grow, more people seek to modify the Linux kernel's MM subsystem to implement these systems. For example, Meta's TPP kernel patch made changes to the NUMA and page reclamation policies to implement a tiered memory system [33] with changes to 22 kernel files. However, Linux's MM subsystem makes it more difficult to add software support for new memory hardware: it is monolithic, and functionality is distributed across dozens of files, many of which require modifications for each extension. In addition to ensuring that their code changes are correct, engineers must also ensure that their changes do not break existing MM functionality. Finally, engineers either must work to get changes upstreamed to the mainline kernel, or take on maintenance of their own fork for the lifetime of the system. In comparison, file systems and storage can be extended through the VFS and block layers, drivers through standardized driver interfaces, networking through protocols, and there is recent interest in extensible scheduling [9]. These components can all be implemented as standalone components without modifying core kernel code. In fact, Linux's MM subsystem stands out as one of the few major hardware-management subsystems in the kernel that is not easily extensible.

With an increase in memory system diversity and heterogeneity, we believe that operating system memory management *must be made extensible* to cope with the rapid increase in demand. We have four goals for an extensibility interface for MM.

1. **Expressiveness**: an extensibility interface must allow expression of a wide variety of MM behavior.

2. **Transparency**: unmodified applications should be able to use MM extensions.

3. **Control**: advanced applications need to specify memory behavior for specific regions, a la `madvise`.

4. **Non-invasive**: in order to ease adoption, the implementation should not require extensive changes to the existing MM code.

Instead of creating a brand new extensibility interface for MM from scratch, we propose *leveraging the extensibility and MM functionality already provided by the VFS layer*. Developers write MM extensions as file systems, which we call *memory management filesystems* (MFSs), and implement the MM functionality in the callbacks provided by the VFS layer. Memory is allocated by creating a file in the MFS's mount directory and then mapping that file. Memory is freed by unmapping and deleting the memory file. We call this system *File Based Memory Management* (FBMM).

The callback functions provided by the VFS layer allow MFSs to control how MM events, such as page faults, are handled, providing sufficient **expressiveness** for a wide variety of MFSs. For **transparency**, we add a small shim layer to the kernel's memory management system that transparently translates MM system calls like `mmap` into file operations by creating memory-backed files and assigning allocation requests to specific files. Our goal of **control** is achieved as a consequence of basing our system on filesystems, which provide a convenient naming mechanism for different MM implementations. Applications can manually create and map files in the mount directory of the MFS that provides the functionality desired for a specific memory region. Most importantly, because our approach builds on existing VFS callbacks, it is **non-invasive** and requires adding only the shim layer to make the system transparent to applications.

The overhead FBMM adds to an individual MM operation is 8% in the worst case scenario of single-page sized allocations and fractions of a percent in the common case of multi-page sized allocations. We have used FBMM to implement memory managers for tiered memory, bandwidth expansion, and contiguous allocations. Each of those memory managers are implemented as standalone kernel modules without additions to the monolithic kernel MM subsystem.

Our implementation has been uploaded to GitHub[1].

2 Motivation and Related Work

Memory has become a dominant factor in system performance, which has led to a multitude of hardware approaches to improve performance that require operating system support. With larger memory sizes, there has been work on improving huge pages [25, 29, 35] and NUMA policies [3]. Fast RDMA networks inspired a renewed interest in remote/disaggregated memory for clusters [15, 32]. Byte-addressable non-volatile memories and high bandwidth memories spur research into tiered memory systems where frequently accessed data is placed in local memory and less frequently accessed memory is stored in slower memory [4, 11, 20, 38, 40]. CXL's memory expansion capabilities also prompt research for tiered memory systems [33] as well as for systems that pool memory between machines [26]. Researchers have also proposed hardware that requires MM changes, such as a TLB that caches contiguous VA to PA translations of arbitrary sizes [22], or hardware to support disaggregated memory [16].

[1]https://github.com/multifacet/fbmm

Problem. Implementing support for new memory hardware generally requires extensions to operating system memory management policies to support new tradeoffs (e.g., near vs. far memory, small vs. large pages) and mechanisms (migration for NUMA, compaction for large pages). Such changes are often difficult to make. They require intimate knowledge of the MM system to locate all of the places in the code that need to be changed, in addition to knowledge of complex data structures and locking patterns. Unlike many parts of Linux and other OS kernels, the MM subsystem is generally monolithic and lacks extensibility.

An example of the complicated and tangled MM code in the Linux kernel is transparent huge pages [2]. The implementation of transparent huge pages is spread across 18 files in the Linux MM subsystem. This code touches a wide range of MM components, such as page fault handling, physical memory allocation, page table management. Additionally, because transparent huge page policies are distributed throughout the MM subsystem, there is an increased likelihood of pathological long latency behavior [29]. Another example is Meta's Transparent Page Placement (TPP [33]), which modifies the NUMA system to support tiered memory. Despite leveraging the existing NUMA code to handle complicated operations like page migration, the authors still modified 22 files in their implementation [31].

Table 1 lists the breadth of changes a selection of recent projects to support better memory management made to the kernel to implement their designs. We organize these changes into sections that represent the core responsibilities of a memory management system: virtual memory management, physical memory management, and translation (e.g., page table management). Adding support for one of these may not be an issue; however, with new memory hardware, we expect there will be many different memory configurations that will need kernel support. Each addition to the monolithic MM subsystem will add to its complexity, making it harder to maintain and expand upon in the future. As such, an extensible interface for MM is imperative to sustain the innovation the boom in new memory hardware promises to bring.

System	Target Hardware	Virtual MM	Physical MM	Translation
TPP [33]	Tiered Memory	N/A	Memory placement decisions. Page migration	Page table updates after migration
Leap [32]	Disaggregated Memory	N/A	Prefetch swapped out pages	N/A
Mitosis [3]	NUMA	N/A	N/A	Replicate and migrate page tables
Range Translations [22]	Range TLB	N/A	Physically contiguous allocation	Manage range based translation table
DVM [18]	Direct Mapping	Make virtual addresses = physical	Physically contiguous allocation	Identity mapping between VA and PA
ASAP [30]	Prefetched Address Translation	N/A	N/A	Page table allocated contiguously

Table 1: Research projects that extend the MM system, and how they extend virtual MM, physical MM, and translation.

Prior work. The interest in extensible MM is not new. In the late 90s, several systems explored this problem [7, 24, 37, 39]. However, the mechanisms for extensibility in these systems focus on application-specific paging policies, rather than extensibility for other MM responsibilities. Likewise, microkernel systems [19, 23, 27, 37] and Exokernels [13, 17] move much or all of memory management out of the kernel to user-mode where it can be extended or replaced. However, these strategies are maximally invasive, as they require whole new kernel designs that make adoption difficult.

More recent work, like HeMem, extend MM by implementing a user library that overloads MM functions like `mmap` using `LD_PRELOAD` [38]. This approach allows extensions to be self contained inside of a library and transparent to the target application, but it lacks the control and information available to a kernel solution, and does not support policies that span multiple applications.

Inspiration. To guide our design of an extensible MM system, we looked at other extensible subsystems. In particular, with the VFS layer, a developer can create a new filesystem by implementing callback functions provided by the VFS layer to perform generic filesystem operations such as open, read, write, etc. Implementations that do not need to modify standard behavior rely on general helper functions, such as `generic_file_open` and `generic_get_unmapped_area` and can implement only a subset of the interface. This greatly simplifies the engineering effort of creating a new filesystem because an engineer only needs to focus on their implementation without having to implement or modify more general filesystem code. As a result, Linux has around 50 filesystem implementations in-tree thanks to the VFS layer. In contrast, Windows has a much lower-level extension interface for file systems [10] and many fewer file system options.

Many filesystems support memory mapping files to an application's memory space, and the VFS layer has callbacks for MM operations, such as page faults, to support this. Linux kernel developers have taken advantage of this in the past for memory management. When support for huge pages was first being discussed, a requirement was that adding support could not overly complicate the existing MM code [28]. This requirement motivated the design of HugeTLBFS, a filesystem-based memory allocator that allows applications to manually allocate huge pages. Because HugeTLBFS is written as a filesystem using VFS, the core of its code is kept in a small set of standalone files. Only a small amount of changes needed to be made to Linux's core MM code, which is what allowed it to be added to the Linux kernel several years before transparent huge pages were supported.

The idea of implementing HugeTLBFS as a filesystem is treated as a one-off solution, and to our knowledge has not been considered again for other systems. It is also not a completely standalone solution. Minor changes have been made to the monolithic MM code to support its use. However, 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.

In order to make VFS a viable extension mechanism for MM, we need to add a layer between the MM syscalls Linux provides and the filesystems that provide the hardware- and policy-specific implementations of those operations for transparency.

3 Design and Implementation

We have the following goals for an extension interface for MM:

1. **Expressiveness**: The extension interface must be able to express a wide variety of MM behavior needed to support modern hardware, such as physical memory allocation, virtual address allocation, and translation management.

2. **Transparency**: A user can change the MM behavior of an application from the default without modifying the

application's source.

3. **Control**: Sophisticated applications can select different MM behavior on different data structures at the same time by explicitly choosing a memory manager for an allocation.

4. **Non-invasive**: The implementation of the extension interface must not overly complicate the existing MM code. Given the existing complexity of OS kernels, we prefer a non-invasive design over a maximally expressive design, so not all extensions may be implementable.

We designed an MM extension interface called *file based memory management* (FBMM) that meets these goals by leveraging the MM capabilities that already exist in the VFS layer. In this design, memory managers are written as filesystems called memory management filesystems (MFS) that implement VFS callback related to MM behavior (e.g. `page_fault`). Mounting an MFS onto the system enables it for use. An MFS can be mounted anywhere on the file path, and our practice is to have one global mount directory accessible by all users for each MFS. Memory is allocated by creating and mapping files in the mount directory of an MFS. We create the *FBMM shim* that transparently replaces an application's dynamic anonymous memory allocations with allocations using a default MFS. After mapping, applications access data with ordinary load/store operations. Applications can also choose which memory manager to use by selecting the mount directory in which to create a files. This enables them to choose the MFS for specific data structures, similar to the use of HugeTLBFS.

3.1 FBMM Overview

Figure 1 FBMM is composed of the *FBMM shim* (described in Section 3.2) and one or more MFSs. The user assigns each process a default MFS (or none) to use on process startup.

In FBMM, the process's syscall interface to MM operations, like the `mmap` and `munmap` system calls remain unchanged. However, with FBMM, whenever a process allocates memory by calling `mmap` with the `MAP_ANON` flag or `brk`, or unmaps memory by calling `munmap`, the MM operation is forwarded to the FBMM shim transparently to the process (1A). Then, the FBMM shim assigns the allocation to a file in the mount directory of the process's default MFS when mapping memory or deletes memory in the file(s) mapped to the memory range when unmapping memory (1B). We call these files *MFS files*. From there, the VFS Layer invokes the process's default MFS, which does whatever processing it needs to do to handle the allocation (1C). Finally, the MFS either allocates the physical memory for the request if `mmap` was called with the `MAP_POPULATE` flag, or frees the physical memory associated with the deleted files if `munmap` was called (1D).

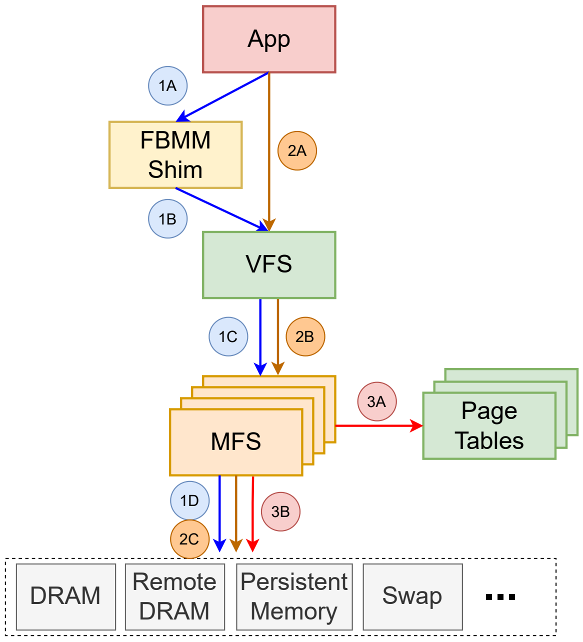

Figure 1: The overall architecture of FBMM.

After a call to `mmap`, the process can read and write to the mapped memory region as it normally would. If the physical memory for the region has not already been populated, the first access will trigger a page fault. Because the faulted memory region is associated with a file, the kernel's page fault handler forwards the fault to the VFS (2A). The VFS, in turn, invokes the page fault handler of the MFS the faulted memory region belongs to (2B). Finally, the MFS allocates physical memory to handle the fault (2C).

MFS operation is not limited to synchronous invocations by the VFS layer. As a part of the kernel, MFSs can spawn kernel threads to perform asynchronous work. For example, a tiered memory system can monitor the hotness of pages by periodically sampling page table access bits inside of an asynchronous thread (3A). With that information, the MFS thread migrates pages itself, without the prodding of the VFS layer or user applications (3B).

The transparency provided by the FBMM shim is important for the ease of use of FBMM; however, some applications may want more control over the MM behavior of specific memory regions. For example, a latency-sensitive application running on a tiered memory system may want a critical data structure to always remain in local memory, regardless of the default MM policy used for the rest of the address space. The application can control allocation by creating and mapping a file in the mount directory of the MFS that provides the desired functionality for the special region. When doing this, the

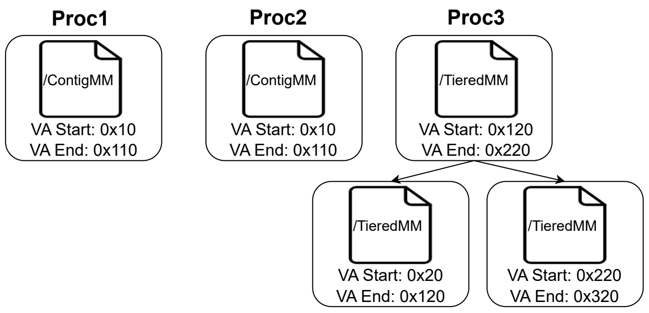

Figure 2: Example of MFS file trees inside of FBMM shim.

application takes on the responsibilities of the FBMM shim for the region. For example, it must create temporary files so they are automatically deleted on process termination, and also set the logical size of the file to at least the desired size of the memory area before mapping, using the `ftruncate` system call. This only sets the logical size of the file, and does not allocate physical memory.

The core Linux MM system still plays an important role in FBMM. It still manages the kernel's private memory and the page cache. Additionally, memory for the stack and BSS/data sections of a process are still the responsibility of the core MM system because having them managed by FBMM would have significantly increased the invasiveness of the system since they are not allocated via `mmap` or `brk`.

3.2 FBMM Shim

The FBMM shim is the mechanism for transparency in FBMM. For each process, it maintains a tree of open MFS files called an *MFS file tree*. When a process calls `mmap` with the `MAP_ANON` flag or `brk`, the kernel invokes the FBMM shim. It calls the `get_unmapped_area` callback of the application's default MFS to get the virtual address range for the allocation. Then, the kernel asks the FBMM shim for a file in the default MFS to map and the page offset to map it to. The FBMM shim searches the process's file tree for an existing file that can satisfy the request. If no such file exists, the FBMM shim creates a new unnamed temporary file and assigns it a compliant virtual address range.

A process's file tree is implemented with the kernel's maple tree implementation and contains every MFS file used by the process. An example of MFS file trees inside of the FBMM shim is shown in Figure 2. Each entry of the file tree contains a pointer to an MFS file and virtual address range the file can be mapped to. MFS files are quite large (discussed below) so many MFS file trees only contain one or two entries, but we do not limit the number of entries in an MFS file tree. Entries are indexed by the start address of their virtual address range.

The naive approach to managing the MFS files would be to have a one-to-one mapping between memory allocations and MFS files. However, there are several issues with this approach. First, creating and opening a new file is expensive, taking about 2-3x the time of a call to `mmap` for anonymous memory. Second, if adjacent memory areas are mapped to different files, the kernel is unable to put those areas in the same VMA structure. This is costly, as allocating a new VMA adds overhead to the `mmap` call, similar to creating and opening a file, and more VMAs makes it more expensive for the kernel to traverse the VMA tree in the future. Additionally, if the process makes many allocations, it is possible to reach the kernel limit of 2^{16} VMAs per process.

The FBMM shim instead shares MFS files across multiple allocations and organizes the allocations within the file to increase the likelihood that they can reside in the same VMA. Other than being in the same file, two memory regions can only be placed in the same VMA if: their permissions are the same, their virtual addresses are contiguous, and the areas they map in the file are contiguous. The FBMM shim does not have control over the first two. The permissions are determined by the caller of `mmap`, and the virtual addresses are controlled by the MFS, though existing implementations of `get_unmapped_area` provided by the kernel, like `generic_get_unmapped_area_topdown`, provide virtual contiguity. However, the FBMM shim can control where allocations reside in a file.

When a new MFS file is created, the FBMM shim sets its logical size to large value (128GB in our implementation) so it can fit many allocations while allocating from the top down. Setting the logical size of a file simply modifies the file's directory entry metadata, so setting the logical size of a file to a large value is no more expensive than setting it to a smaller value. In order to ensure that allocations maintain the same relative placement inside the file as they do in virtual memory, the MFS associates each file with a virtual address range of equal size to the file. When the FBMM shim is asked to provide an MFS file for a new allocation, it searches through the process's file tree for a file such that the allocation is wholly within the file's virtual address range. If such a file is found, the FBMM shim returns it for the allocation to map. The FBMM shim assigns the allocation an offset into the file equal to the allocation's start address subtracted by the start of the file's virtual address range. The FBMM shim does not need to track which regions of a MFS file are allocated because `get_unmapped_area` implementations cannot allocate overlapping virtual address regions.

If a suitable file is not found, the FBMM shim creates a new one. The virtual address range of the new file is determined by the virtual address of the allocation and whether the allocation is done via `brk` or `mmap`. If the allocation was done by `brk`, the start of the file's virtual address range is set to the start address of the allocation because `brk` grows the address space upwards. If the allocation was done by `mmap`, the end of the file's virtual address range is set to the end address of the allocation because `mmap` typically grows the address space downwards.

When a process calls munmap, the FBMM shim searches the process's MFS file tree and punches a hole into each file that overlaps with the range of addresses being unmapped by calling fallocate with the FALLOC_FL_PUNCH_HOLE flag on the files. Similarly, when a process terminates, the FBMM shim traverses each entry of the process's file tree and deletes each file. These two actions signal to the corresponding MFS that physical memory should be freed.

We integrate the FBMM shim into the kernel, but an alternative implementation would be to create it as a userspace library that intercepts processes' calls to MM related glibc functions via LD_PRELOAD, similar to the implementation of HeMem [38]. We chose to pursue a kernel solution because a userspace implementation would incur the overheads of crossing the kernel boundary for each file operation in addition to the MM system calls. Additionally, while unlikely, applications can invoke MM system calls directly rather than going through the glibc functions, bypassing a userspace solution. For these reasons, we believe the kernel implementation is a more robust design, despite the need for modest kernel changes.

3.3 MFS Design

An MFS can be implemented with only a subset of the callbacks provided by the VFS layer. A list of particularly important callbacks are listed in Table 2. With these interfaces, an MFS is able to control how virtual addresses are allocated (get_unmapped_area), how physical memory is allocated (page_fault and fallocate), and how physical memory is freed (free_inode and fallocate). An MFS can also be signaled when a file is mapped for the first time (mmap).

In addition to the interfaces from the VFS layer, MFSs have access to interfaces available to other kernel subsystems because they themselves are parts of the kernel. They can allocate physical pages directly by statically reserving memory at boot time, or can dynamically allocate and free physical memory by calling the alloc_pages and put_page kernel functions. MFSs can also traverse process VMA trees and modify page tables.

Because the core of their implementation is done as callback functions, MFSs can be written as completely standalone pieces of software, like most filesystems. In fact, an MFS can be created as an independent kernel module that is compiled and loaded separately from the main kernel.

3.3.1 Virtual Memory Management

The primary way an MFS controls the virtual addresses of an allocation is via the get_unmapped_area VFS callback. The get_unmapped_area callback passes the length of the allocation, the mmap flags it was called with, and an address hint provided by the caller to the MFS. Then, the MFS finds a suitable virtual memory region in the caller's address space that satisfies the input parameters as well as the MFS's design goals. For example, an MFS implementing devirtualized memory [18] would have the virtual addresses be equal to the physical addresses used for the allocation. MFSs could also use virtual addresses to encode information about the allocations as in OVC [6]. In order to minimize the overhead of the FBMM shim, implementations of get_unmapped_area should allocate virtual addresses contiguously and grow the address space downward to allow for VMA merging (as detailed in Section 3.2).

MFSs that are not particular about virtual addresses selection can point the get_unmapped_area callback to existing helper functions in the kernel, like generic_get_unmapped_area_topdown and thp_get_unmapped_area.

3.3.2 Physical Memory Management

There are two ways an MFS is alerted that a process needs physical memory: the page_fault VFS callback and the fallocate VFS callback. The kernel invokes page_fault callback during a page fault and gives the MFS the address that triggered the fault. The fallocate callback, which is used to tell filesystems to preallocate disk space for a file, is invoked by the FBMM shim when mmap is called with the MAP_POPULATE flag and gives the MFS the memory range to allocate memory for. In both callbacks, the MFS can make decisions such as where the physical memory should be allocated from and whether or not to use huge pages.

Similarly, there are two ways an MFS is alerted that a process's memory can by freed: the free_inode callback and the fallocate callback when it is called with the FALLOC_FL_PUNCH_HOLE flag. The FBMM shim invokes the free_inode callback when an FBMM file is deleted, which occurs after the process using the file terminates, and tells the MFS to free all of the physical memory belonging to that file. When a process calls munmap on a region including an FBMM file, the FBMM shim invokes the fallocate callback with the FALLOC_FL_PUNCH_HOLE flag, which gives the MFS a memory range to free.

Allocating and freeing physical memory is not limited to these callbacks. For example, for a tiered memory MFS to migrate a page from the local node to the remote node in a kernel thread, it would need to both allocate a page in the remote node and free a page in the local node. Additionally, some MFSs, such as those used to implement RMM [22] and DVM [18], may want to always preallocate physical memory for a region whether or not the region was created with the MAP_POPULATE flag. This can be accomplished by allocating physical memory at the same time virtual memory is allocated in the get_unmapped_area callback.

MFSs need to know what parts of MFS files are backed by physical memory and what pages they are backed by so those pages can be freed later. Traditional filesystems solve

Interface	Defined in	Called by	Purpose
`mmap` (callback, not syscall)	`struct file_operations`	`mmap` syscall	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
`fallocate`	`struct file_operations`	`mmap` syscall with `MAP_POPULATE` flag / `munmap` syscall	Signal need to allocate / free physical memory
`fault`	`struct vm_operations_struct`	Page fault handler	Control the paging behavior of a process
`free_inode`	`struct super_operations`	File deletion code	Signal need to free physical memory

Table 2: Interfaces used by MFSs.

a similar problem by managing indexing structures that map file offsets to disk blocks. Such a structure is not generally needed in an MFS because they can walk the page tables of the processes that map their files to get this information.

An MFS needs access to physical pages in order to assign them to processes. This access can be granted statically by reserving a chunk of memory at boot time or immediately when the MFS is mounted. An MFS can also allocate physical pages dynamically using standard kernel functions like `alloc_pages`. Both strategies have their benefits.

Reserving physical pages statically guarantees that a certain amount memory will be available to an MFS without having to worry about the memory usage of the rest of the system. It gives the MFS more control over the memory. For example, a problem that some systems experience is interference from internal memory fragmentation [29]. By statically reserving a block of pages, an MFS can guarantee that specific contiguity requirements are met. The control of a static reservation can also help simplify physical page allocation, leading to performance improvements. With statically allocated pages, an MFS can use data structures other than the kernel's buddy heap, such as a simple free list of base pages for speed, or a tree of free segments for contiguous allocation.

Dynamically reserved pages have the benefits of flexibility. When an MFS reserves pages dynamically from the kernel, it eliminates concerns of overprovisioning physical pages, taking away resources from the rest of the machine needlessly, or underprovisioning and not having enough pages to satisfy the requests of the applications using it. MFSs can also respond to kernel memory pressure by registering a "shrinker" callback with the kernel. Shrinkers are typically used by the kernel to tell drivers to free memory by clearing their caches, but MFSs can use it as a signal that it needs to start returning physical memory to the kernel [8]. Additionally, by using built in kernel functions to allocate pages, the MFS does not have to implement its own page allocator.

Regardless of whether an MFS reserves physical pages statically or dynamically, MFSs will still generally manage physical pages via the kernel `page` and `folio` structures.

3.3.3 Virtual to Physical Translation

MFSs handle virtual to physical translation by modifying the page tables of the process's using them. Translation in MFSs are primarily created in the `page_fault` VFS callback. In most cases, this involves simply populating PTEs in the page table, but the `page_fault` callback is also the natural place to implement alternative page tables designs, like those proposed in RMM [22] and Mitosis [3]. Existing kernel helper functions, such as `mk_pte`, which creates a PTE entry from a physical page and access permissions, and `walk_page_range`, which walks a process's page table invoking callbacks provided by the caller for each entry, help MFSs accomplish these translation tasks. While most translation work occurs in the `page_fault` callback, an MFS can traverse and edit a process's table at any time. This is useful, for example, to periodically monitor page table access and dirty bits in a kernel thread.

3.4 Discussion

This design satisfies our four goals for an MM extension interface. The VFS layer's support of memory mapped files for traditional filesystems along with MM helper functions available in the kernel provide an **expressive** interface that allow engineers to express a wide variety of MM behaviors (see Section 5). The FBMM shim allows processes to use FBMM **transparently** by translating standard MM functions to file operations in the default MFS's mount directory. Furthermore, the default MFS is chosen on process start without any change necessary to application code. If an application wants **control** over the MM behavior of specific memory regions, it can manually create and map files in the mount directory of the MFS that provides the desired behavior. Finally, piggybacking off of the kernel's longstanding ability to memory map files into a process's address space allows FBMM to be **non-invasive** to the existing MM code - only modest changes were needed to invoke the FBMM shim (see Section 3.5).

Impact on kernel MM With FBMM, we envision that most MM policy decisions will be moved inside MFSs. As such, we believe the role of the core kernel MM subsystem should be providing a sound foundation for the MFSs to build upon. This includes things it already does well, such as forwarding MM events to MFSs and doing bookkeeping needed for most MM implementations, such as maintaining the VMA list and creating default page tables. A solid foundation also involves providing useful helper functions to MFSs for common MM operations, acting as a software library to more easily create MFSs. Some useful helper functions, like `alloc_pages` for allocating physical memory and `walk_page_range` for walking the page table, are already available to MFSs. We have also created helper functions to help MFSs handle swapping and copy-on-write. As more MFSs are written, we believe it will become clear what other helper functions would be useful. Finally, a simple memory manager should remain in the core MM code for the kernel to use, to manage stack and data/BSS segment memory, and to bootstrap the system on startup.

Limitations There are some limitations to the design of FBMM. The first is the MFSs are not composable. If, for example, one had an MFS for huge pages and another for tiered memory, they cannot be "stacked" together to make an MFS for tiered memory using huge pages. A second limitation is there is no easy way for multiple MFSs to coordinate with each other. For example, such functionality could be useful to decide which MFS should be chosen as a victim to swap out pages under memory pressure. Additionally, previous research has shown benefits in extending the MM behavior of kernel processes [21]. Sadly, using FBMM on the kernel is not supported as the kernel cannot memory map files. For the same reason, FBMM cannot be used to manage page cache memory. Similarly, FBMM does not apply to memory in the stack or data/BSS segments because these regions are not allocated with `mmap` or `brk`. However, these memory sections are often much smaller than dynamically allocated regions, so we believe they do not require as much specialized behavior. Furthermore, an MFS may not support all the features of Linux anonymous memory, such as the handling of copy-on-write after a process is forked, so allocations or operations may fail for applications that use these. Finally, an MFS implementation must fit inside the framework provided by the VFS layer. For example, Mitosis [3] replicates a process's page table on each NUMA node and modifies the context switch code to decide which page table to use. This is not possible to do inside of an MFS alone as the VFS does not have any callbacks for context switches. However, it would still be beneficial to implement the page table replication in an MFS after the other changes are made directly to the kernel.

3.5 Implementation

We implemented FBMM in Linux kernel version 6.2. The FBMM shim was implemented in about 600 lines of code. Changes to the rest of the kernel MM code were minimal; only about 50 lines of code were added to interface with the FBMM shim where appropriate.

In our implementation, a process's default MFS is set by writing its mount directory to `/proc/<pid>/FBMM_mount_dir`. For our experiments, we use a wrapper application that sets the default MFS and then calls `execv` to run the desired application.

In addition to the kernel changes to support FBMM, we have also implemented four different MFSs to demonstrate FBMM's extension capability. These include a bare-bones MFS that simply allocates base pages, and MFSs for tiered memory, bandwidth expansion, and contiguous memory allocation. They are described in more detail in Section 5.

4 Performance Evaluation

FBMM, like any other abstraction layer, comes with some level of overhead. The main source of this overhead is in the creation and management of files in the FBMM shim.

Applications only invoke FBMM shim when a memory region is mapped/unmapped; subsequent callbacks go directly to the VFS layer. Therefore, to stress the FBMM shim, we created a microbenchmark that calls the `mmap` system call multiple times in quick succession in one or more threads. We also instrumented the FBMM shim to measure the time spent managing files when mapping/unmapping a region.

All experiments in this paper are run on bare metal Cloudlab [12] c220g1 machines with two Intel E5-2630 v3 8-core CPUs and 128GB of ECC DDR4 RAM spread across two NUMA nodes. We set the CPU scaling governor to `performance`, fixing the clock frequency to 3.2GHz. Due to the lack of CXL hardware, we approximate remote memory as memory accesses to the remote NUMA node.

BasicMFS We implemented a simple bare bones MFS we call BasicMFS to measure the minimum cost of an MFS. When it is first mounted, BasicMFS pre-reserves the pages it will use for its operation from the kernel and places them in a linked list of free pages. It defers to the kernel to allocate virtual address space. When memory is requested via the `page_fault` or `fallocate` VFS callbacks, it allocates a 4KB base page by popping one from the free list to satisfy the request. When memory is freed, such as via the `free_inode` callback, BasicMFS places the freed pages back onto the free list. BasicMFS is implemented in about 550 lines of code. This simple design, while not useful in practice, is helpful for measuring the minimum cost of a MFS and for understanding the minimal implementation of a MFS.

FBMM shim Latency. We first measure how the latency overhead of FBMM scales with the size of the allocation compared to base Linux. We configure the microbenchmark to measure the latency of `mmap` with the `MAP_ANON` flag and without the MAP_POPULATE flag 100,000 times with allocation sizes ranging from 1 to 128 pages, and measure the latency of calling munmap on those regions. We use BasicMFS for these experiments. Table 3 shows the results.

These results show that time it takes to call mmap/munmap is near constant as the allocation size varies when using both base Linux and FBMM, which is expected because the operation only reserves virtual address space but does not allocate and zero memory. However, calling mmap is about 15-20% slower with FBMM than with base Linux. Similarly, calling munmap is between 30-40% slower than Linux. This overhead comes from the FBMM shim's work to manage the file tree and locate a file for the allocation. With the MAP_POPULATE flag, the time spent in the FBMM shim when allocating a single page is equal to 8% of the time base Linux takes to allocate a page. This overhead decreases proportionally with the allocation size, down to 0.1% of the cost of allocating 128 pages. When allocation is included, the overhead introduced by the FBMM shim is overshadowed by the cost of allocating and zeroing free pages. In fact, as shown in the columns marked "Populate" in Table 3, FBMM with BasicMFS is between 4-8% faster than Linux at mmap with the MAP_POPULATE because of its simpler page allocation path. Likewise, BasicMFS is up to 45% faster than Linux at munmap when the memory region is populated because of the simple page freeing path. Results where the microbenchmark touches pages to trigger allocation during a fault, rather than with MAP_POPULATE, behave similarly.

We also measure how FBMM scales with an increasing number of threads allocating memory at once. We use the same configuration as above, with 1-page allocations while varying the number of threads calling mmap from 1 to 32. This stresses the FBMM shim code walking the process file tree. The overhead of FBMM remains between 15-20%, similar to the last experiment, regardless of the number of threads used. This makes sense because the mmap system call serializes these requests with the mmap_lock.

Extrapolation to applications. Our microbenchmark stresses the performance of the FBMM shim, but is not representative of how applications use memory, as user-mode heaps typically allocate large chunks of memory infrequently and then satisfy small memory requests from these chunks. To understand real-world behavior, we measured the frequency of mmap and brk calls made by various applications. These results are listed in Table 4. Calls to mmap and brk are relatively infrequent, typically less than twice a second, which makes the overhead of FBMM shim negligible. Furthermore, each call to mmap/brk typically allocates several thousand pages, amortizing overheads further. The exception to both of these is canneal, which on average makes an allocation of 54 pages twenty times a second for an overhead of a 0.23% (a few microseconds delay per second). Additionally, the cost of allocating tens of physical pages is still enough to further hide overheads.

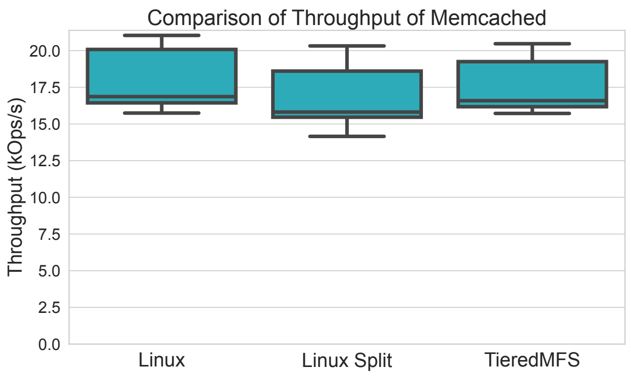

Figure 3: Stem-and-whiskers plot of the average throughput over 50 executions of Memcached driven by YCSB in a read only workload in the Linux, Linux Split, and TieredMFS configurations.

Once FBMM allocates a physical page and maps it into a process page table, the cost of load and store instructions in those regions will be the same as loads and stores to regions managed by the kernel's default memory manager, as there is no software overhead. Differences in performance at this point occur due to policies and mechanisms of the specific MFS, such as manipulations of the page table (e.g., clearing access and dirty bits, TLB flushes) or page migration.

5 Case Studies

In addition to the BasicMFS, we implemented three MFSs inspired by previous works on memory management to show the power of FBMM. Unlike their original implementations, each MFS is a standalone kernel module built and loaded separately from the main kernel.

5.1 Tiered Memory

We have built a tiered memory MFS based on the design of TPP [33] called TieredMFS. The goal of TieredMFS is to place a process's frequently accessed (hot) data in faster local memory, while placing less frequently accessed (cold) data in slower remote memory. It is implemented in about 1500 lines of C code, about a third of which is for debugging and boilerplate for defining a filesystem.

The memory available to TieredMFS is segregated into local and remote memory pools. These memory pools are reserved statically at boot time for simplicity. Each pool keeps a list of hot pages and a list of cold pages allocated from that pool. Periodically, a kernel thread samples the page table accessed bits of allocated pages and adjusts the hot and cold lists accordingly. Similarly, a kernel thread will periodically monitor the amount of memory available in the local pool. If

System	1 page	2 pages	8 pages	32 pages	128 pages
Linux	0.78 / 1.50	0.70 / 1.53	0.71 / 1.54	0.69 / 1.49	0.70 / 1.50
FBMM	0.94 / 2.05	0.83 / 1.96	0.83 / 1.99	0.82 / 2.02	0.84 / 2.01
Linux - Populate	2.17 / 2.36	3.16 / 2.55	9.56 / 4.01	35.18 / 9.34	136.99 / 22.50
FBMM- Populate	2.08 / 2.50	2.97 / 2.62	8.73 / 3.74	31.52 / 7.28	125.21 / 14.88

Table 3: Average time spent to `mmap`/`munmap` in microseconds.

Application	`mmap` calls	`brk` calls	Average Size (Pages)	Allocation Frequency (Hz)
xz	538	6	22,000	1.2
mcf	47	7	45,000	0.06
cactuBSSN	195	11	8500	0.5
canneal	9	4,691	54	20
Memcached	930	3	17,000	1.8

Table 4: Breakdown of `mmap` and `brk` calls made by applications.

the available memory is below an administrator defined reclamation threshold, pages from the bottom of the local pool's cold list will be migrated to the remote pool and placed on the top of the remote pool's cold list. If the available memory is above an administrator defined allocation threshold, which is lower than the reclamation threshold, pages from the remote pool's hot list will be migrated to the local pool and placed on top of the local pool's hot list.

When a process using TieredMFS requests a physical page, it is placed in the local pool if the available memory is above the allocation threshold. Otherwise, it is placed in the remote pool. In both cases, the page is placed on the hot list of the pool it is allocated to. This logic is implemented inside of the VFS `page_fault` callback. TieredMFS supports mapping 2MB huge pages as well as 4KB base pages. However, we have found that the smaller base pages are more useful for determining hot regions, so we use base pages for our experiments.

We evaluated TieredMFS with a modified version of the GUPS microbenchmark where 90% of accesses go to a hot region of memory, and the addresses that make up the hot region change partway through execution (originally used with HeMem [38]). We run GUPS with 32GB of data and a hot region size of 8GB, and configure TieredMFS with 8GB of local memory and 64GB of remote memory, and compare against standard Linux's default NUMA policy with the same local/remote allocation (Linux Split), and standard Linux where all of the workload's memory fits comfortably in local memory (Base Linux). Regrettably, we were unable to get TPP working (and confirmed others experienced similar problems), so we cannot compare it to the performance of TieredMFS.

Table 5 shows the performance of Linux Split and TieredMFS relative to the performance of Base Linux. TieredMFS outperforms Linux Split because it lowers the number of memory accesses going to remote memory. It does this by identifying the new hot set when the access pattern changes and demotes no longer hot pages to remote memory and migrates the new hot set to local memory. Linux Split on the other hand is hurt by the fact that NUMA's ability to demote memory is limited, leaving no room for the new hot set to enter local memory [33].

System	Relative Throughput	Remote Access %
Linux Split	70%	20%
TieredMFS	88%	6.5%

Table 5: Throughput of GUPS as a percentage of Base Linux throughput and the percentage of memory reads going to remote memory. We only measure reads because there is no perf counter for stores that miss the last level cache.

We also evaluated TieredMFS's performance with Memcached using YCSB with a read-only zipfian workload using the same configurations as above. Because the throughput results vary, we ran the experiment fifty times per configuration and report the results in Figure 3. The median throughput when using TieredMFS is 98% of the median throughput of Base Linux, while Linux Split achieves only 94% of Base Linux. Again, the performance gains of TieredMFS over Linux Split is due to the number of remote memory accesses. With Linux Split, 57% of memory reads are served by remote memory, compared to less than 0.5% in TieredMFS.

5.2 Bandwidth Utilization

SK Hynix recently released the HMSDK library and modified kernel that supports heterogeneous memories [1]. A key feature is interleaving a process's memory across the memory nodes of the machine to maximize the memory bandwidth available to the process. We implemented the same functionality as an MFS called BWMFS. With BWMFS, an administrator sets the allocation weights for each memory node via a sysfs interface created when BWMFS is mounted: a local:remote weight ratio of 3:2 indicates 60% of memory should be allocated locally and 40% remotely. When a pro-

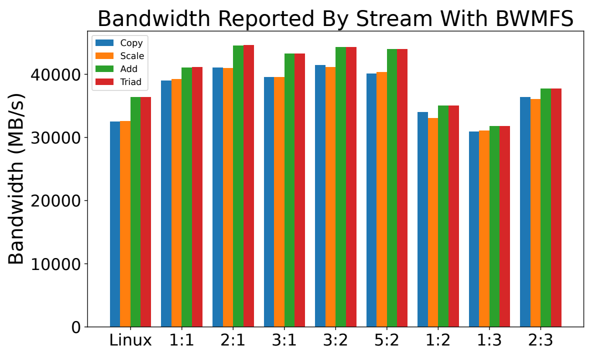

Figure 4: Memory bandwidth results calculated by the STREAM benchmarks with Linux and BWMFS with different local:remote allocation ratios. 2:1 means 2 pages are allocated to local memory for every 1 page allocated to remote memory.

cess requests physical memory, a BWMFS will allocate a page chosen from one of the available nodes in a round robin fashion, weighted by the provided allocation weights. This allocation is done inside of the `page_fault` and `fallocate` callbacks which use the `alloc_pages_node` kernel function to allocate physical pages dynamically from the desired node. BWMFS currently only supports allocating base pages, but we plan on adding huge page support. Because of the extensibility of FBMM, this prototype bandwidth expanding memory manager was able to be written in a single afternoon.

We test the functionality of BWMFS by using it as the default MFS for a version of the STREAM benchmark [34] running with 8 threads to saturate the bandwidth on the local node. The original STREAM benchmark exercises memory bandwidth by accessing global variables, so we modified the benchmark to allocate those variables via `mmap` instead. Figure 4 shows the bandwidth results of running STREAM with Linux and BWMFS with various local:remote node allocation ratios. BWMFS with allocation ratios of 1:1, 2:1, 3:1, 3:2, and 2:3 outperform Linux's MM which only allocates to the local node because BWMFS is able to utilize more bandwidth across the two nodes than it can with the local node alone.

We ran the same experiments with HMSDK to see how BWMFS compares. In all cases, the results from BWMFS are within ±3% of the results from HMSDK.

5.3 Contiguous Allocation

Redundant Memory Mappings proposes adding a software "range page table" with a corresponding hardware "range TLB" to CPU translation hardware that caches virtual to physical address mappings of arbitrarily sized memory ranges contiguous in both virtual and physical memory [22]. To increase the effectiveness of the range TLB, the authors modify

	% TLB Misses Prevented
mcf	99.78%
cactuBSSN	99.92%
GUPS	99.91%

Table 6: Percentage of an application's TLB misses prevented by using ContigMFS as the default MFS, with a 32 entry range TLB

System	Files Changed	Lines Changed
TPP [31]	22	471
TieredMFS	3	1567
HMSDK [1]	9	920
BWMFS	2	579
RMM [22]	16	546
ContigMFS	2	479

Table 7: The number of files and lines of code needed to implement MFSs and the systems they were based on.

the MM code to eagerly allocate physical memory when `mmap` is called, rather than lazily allocating physical memory on the first use of each page. This increases the contiguity of the physical memory, allowing a larger range to be cached in the range TLB. We have implemented these extensions to MM as an MFS called ContigMFS.

To eagerly allocate a memory region's physical memory, ContigMFS allocates all of its physical memory inside the `get_unmapped_area` VFS callback. ContigMFS allocates contiguous blocks of physical pages using the `folio_alloc` kernel function. This design choice simplifies the implementation of ContigMFS, allowing it to piggyback off of the kernel's existing capability for contiguous allocation rather than creating its own implementation. Since it has already allocated the physical memory at this point, ContigMFS also populates the relevant page table entries, as well as the entries for the range page table inside of the `get_unmapped_area` callback. The ContigMFS is implemented in only 479 lines of code.

Because no available hardware implements RMM, we simulate it inside of the kernel with a modified version of BadgerTrap [14] that counts the total number of page table walks that occur in a process along with the number of walks that would have been prevented by a range TLB, as is done by the authors of RMM, in order to test the functionality of ContigMFS. Table 6 shows the percentage of TLB misses that would be prevented in the selected applications when using ContigMFS with a 32 entry range TLB compared to just using base pages. ContigMFS's ability to allocate large regions of physical memory contiguously and populate the novel range tree allows applications to reduce the number of TLB misses they suffer dramatically.

5.4 Discussion

Each of the MFSs described in this section are able to express complex MM behavior and maintain competitive performance while being implemented as standalone kernel modules. Writing memory managers as kernel modules with FBMM helps simplify their implementation. This is shown in Table 7. Systems like TPP, HMSDK, and RMM must spread their implementations across many files and require the implementors to have a good understanding of the complicated code they build their policies on top of. On the other hand, the implementations of TieredMFS, BWMFS, and ContigMFS only span a handful of independent files.

These modular implementations are not always smaller than monolithic additions. TieredMFS adds 3x more lines of code than the implementation of TPP (2x if you discount boilerplate and debug code) because TieredMFS implements page migration and hotness tracking itself, while TPP integrates itself into the NUMA subsystem to accomplish those tasks. However, we believe the benefits of writing memory managers as standalone pieces of software that do not further add to the kernel's technical debt outweigh the costs of occasionally re-implementing behavior. Furthermore, better kernel abstractions such as DAMON [36] for hotness tracking could reduce the implementation size.

Part of the reason the MFS implementations are only hundreds of lines of code is that they do not need to support every standard MM feature of anonymous memory. An MFS is not required to support huge pages, and the same goes for functionally specific MM features like copy-on-write. As a result, when using an MFS as an application's default memory manager, it is important to ensure the MFS supports all of the features the application requires to run.

6 Conclusion

New MM policies are needed to effectively make use of the explosion of new memory hardware in recent years, such as CXL, high bandwidth memories, and persistent memories. However, the kernel MM subsystem's monolithic design makes adding the policies challenging, and those additions further complicate the monolith. FBMM solves this problem by leveraging the VFS layer's MM capability, allowing memory managers to be written as standalone kernel modules as filesystems (MFSs). The FBMM shim allows users to choose the MFS to use for an application transparently to the application by intercepting MM syscalls and translating them to filesystem operations in the MFS. The overhead FBMM adds to allocating memory ranges between 8% when allocating a single page, and quickly decreases to near zero as the number of pages allocated increases.

Acknowledgements

We thank Mark Mansi for his contributions to the early parts of this project. We thank the anonymous reviewers, Sujay Yadalam, and Anjali for their time and feedback on our paper. We thank the anonymous artifact reviewers for their time testing our artifact.

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] HMSDK v1.1 Design. `https://github.com/skhynix/hmsdk/wiki/HMSDK-v1.1-Design`.

[2] The Linux Kernel User's and Andministrator's Guide: Transparent Hugepage Support. `https://www.kernel.org/doc/html/next/admin-guide/mm/transhuge.html`.

[3] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020.

[4] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[5] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient Virtual Memory for Big Memory Servers. *SIGARCH Comput. Archit. News*, 2013.

[6] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Reducing Memory Reference Energy with Opportunistic Virtual Caching. *SIGARCH Comput. Archit. News*, 2012.

[7] 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.

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

[9] Jonathan Corbet. The Extensible Scheduler Class. https://lwn.net/Articles/922405/, 2023.

[10] Microsoft corp. File systems and minifilters. https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/_ifsk/.

[11] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, 2016.

[12] 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, 2019.

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, 1995.

[14] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. BadgerTrap: A Tool to Instrument X86-64 TLB Misses. *SIGARCH Comput. Archit. News*, 2014.

[15] 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.

[16] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, 2022.

[17] 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.

[18] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, 2018.

[19] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The Performance of μ-Kernel-Based Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, 1997.

[20] Taekyung Heo, Yang Wang, Wei Cui, Jaehyuk Huh, and Lintao Zhang. Adaptive Page Migration Policy With Huge Pages in Tiered Memory Systems. *IEEE Transactions on Computers*, 2022.

[21] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. KLOCs: Kernel-Level Object Contexts for Heterogeneous Memory Systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, 2021.

[22] 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, 2015.

[23] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[24] 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.

[25] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and Efficient Huge Page Management with Ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA, November 2016.

[26] 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 '23, 2023.

[27] Jochen Liedtke. On Micro-Kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, 1995.

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

[29] 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.

[30] 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.

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

[32] Hasan Al Maruf and Mosharaf Chowdhury. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference*, USENIX ATC '20, 2020.

[33] 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 '23, 2023.

[34] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995.

[35] Ashish Panwar, Sorav Bansal, and K. Gopinath. HawkEye: Efficient Fine-Grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, 2019.

[36] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. Profiling dynamic data access patterns with controlled overhead and quality. In *Proceedings of the 20th International Middleware Conference Industrial Track*, page 1–7, 2019.

[37] 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.

[38] 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.

[39] 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.

[40] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, 2019.

[41] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial Failure Resilient Memory Management System for (CXL-Based) Distributed Shared Memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, 2023.