12-2018

Low-Overhead Migration of Read-Only and Read-Mostly Data for Adapting Applications to Hybrid Memory Systems

Joseph Townley Teague
University of Tennessee

To the Graduate Council:

I am submitting herewith a thesis written by Joseph Townley Teague entitled "Low-Overhead Migration of Read-Only and Read-Mostly Data for Adapting Applications to Hybrid Memory Systems." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael Jantz, Major Professor

We have read this thesis and recommend its acceptance:

Micah Beck, Max Schuchard

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Low-Overhead Migration of Read-Only and Read-Mostly Data for Adapting Applications to Hybrid Memory Systems

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Joseph Townley Teague

December 2018

This thesis is dedicated to:

Mom and *Dad*, for always encouraging me to strive onward

Sarah, for **always** being there while I was working and never letting me give up

Mike, for giving me the confidence, instruction, and time to accomplish this work

Liz and *Joe*, for being so interested in what I've been doing this whole time

and

Isabella, for giving me a reason to do more with myself

Acknowledgments

I would like to thank my advisor, Dr. Michael Jantz, at the University of Tennessee, Knoxville for supervising me while I worked on this. Mike's expertise, patience, and encouragement helped me see this through to the end while allowing me to develop my personal skills and explore in my own direction (within reason, of course).

I would also like to extend my thanks to the other two members of my committee, Drs. Micah Beck and Max Schuchard, for being part of this important step in my education. While I did not call on them as frequently as my advisor, they too were never unavailable.

At Intel, I would like to thank Drs. Kshitij Doshi and Suleyman Sair for the tutelage they provided. They both took time away from their important work on a regular basis to instruct me on new topics, and my time in Arizona would not have been as personally or professionally formative as it was without them.

At the University of Tennessee, Knoxville, I would like to thank Drs. Micah Beck, Michael Jantz, Gregory Peterson, and Jim Plank for the excellent architecture and systems courses they taught. These classes were fundamental to the knowledge that allowed this thesis to become a reality. Drs. Michael Berry, Bradley Vander Zanden, and Steven Marz deserve my thanks for letting me be their teaching assistant from time to time and furthering my love of teaching.

Within UT's Compilers, Operating, and Runtime Systems (CORSys) group, I would like to thank my lab-mates Ben Olson, Adam Howard, Chad Effler, Divyani Rao, Tong Zhou, Tasmia Rahman, Rayhan Hossain, and Brandon Kammerdeiner for the stimulating conversations, comic relief, and validation of my work.

Finally, within my family, I would like to thank my mother and father, Victoria Medaglia and Tom Teague, for the neverending encouragement (and for helping me edit this

document), my fiancée Sarah Schaffer for the constant support and understanding during the final, time-consuming months of writing this thesis, and my daughter Isabella Teague for being the ultimate inspiration to further my education.

Abstract

Memory systems containing different types of memory with varying capacity, latency, and bandwidth are rapidly becoming mainstream. Conventional memory management techniques do not suffice for these systems; they require alternative strategies to appropriately and effectively adapt application memory placement to these heterogeneous memory tiers. Software-based placement and movement strategies are the most desirable due to their flexibility and ease of adoption by end-users. However, there are substantial sources of overhead present when synchronizing low-level data movement with the operating system and running applications.

This thesis proposes a novel method of reducing these memory movement overheads on hybrid memory systems. Many data objects are only written to early in their life cycle (i.e. shortly after allocation) and are effectively read-only after these initial writes. If this read-only and read-mostly data is duplicated across memory tiers, as opposed to moved, the application, in many cases, is able to avoid certain types of transfer overhead, such as page table entry (PTE) and MMU cache (TLB) synchronization stalls.

This work describes the design and implementation of a kernel module, `mtier` that implements this optimization on memory that has been explicitly marked as read-only. Our evaluation demonstrates that this approach has the potential to substantially reduce data movement overheads, especially in applications that are multi-threaded and require frequent movement of data, allowing a flexible, software based approach for memory management in hybrid systems.

Table of Contents

1 Introduction **1**

2 Background **4**

2.1 Production Systems 5

2.1.1 Experimental Systems 6

2.2 What's Missing 8

3 The `mtier` Framework **9**

3.1 Principles of Operation 9

3.2 Kernel Modifications 10

3.3 The Helper Program 12

3.4 The Module 12

3.4.1 The `tier_struct` 12

3.4.2 The Linked Lists and the Hash Table 14

3.4.3 Operation 15

3.4.4 Bookkeeping Iterations 15

3.4.5 Standard Iterations 17

3.4.6 Dummy Iterations 19

3.5 Eviction Modes 21

3.5.1 `evict_unused` 21

3.5.2 `evict_pid` 21

3.5.3 `evict_all` 22

3.6 Handling Edge Cases 22

3.6.1 Process Exit . . . 22
3.6.2 Module Exit . . . 23
3.7 Module Experimental Configurations . . . 23
3.7.1 `mtier_heavy` . . . 23
3.7.2 `mtier_massive` . . . 24
3.7.3 `stream_madvise` and `stream_mbind` . . . 24
3.8 Module Development Configurations . . . 25
3.8.1 `mtier_dev_full` . . . 25
3.8.2 `mtier_dev_partial` . . . 26
3.9 Deficiencies . . . 26
3.9.1 Multi-Process Behavior . . . 26
3.9.2 Early Module Exit . . . 26
3.9.3 Mixed Kernel-Module Approach . . . 27

4 Experiments **28**
4.1 Hardware . . . 28
4.2 Benchmarks Used . . . 28
4.2.1 Matrix Multiply . . . 29
4.2.2 STREAM . . . 30
4.2.3 STREAM with Validation . . . 32
4.2.4 Benchmark Comparison . . . 33
4.3 Methodology . . . 33
4.3.1 Baseline Configurations . . . 34
4.3.2 `mtier` Configurations . . . 35
4.3.3 STREAM Configurations . . . 35
4.3.4 The Experiment Script . . . 35

5 Evaluation **37**
5.1 Feasibility . . . 37
5.2 Performance . . . 37
5.2.1 Frequency of TLB Shootdowns . . . 38

5.2.2 Migration and Movement Performance . . . 41
5.3 Effect of Epoch Time on Performance . . . 43
5.4 Effect of Simulated Fast Tier Size on Performance . . . 45
5.5 Effect of Number of Threads on Performance . . . 46
5.6 Analysis of `mtier` . . . 47
5.7 Analysis of `mtier_heavy` . . . 48
5.8 Analysis of Userspace Solution . . . 49

6 Future Work **50**
6.1 Module-Only Framework . . . 50
6.2 Improved Page Eligibility Determination . . . 50
6.3 Write Protection . . . 51
6.4 Managed Language Support . . . 51
6.5 `MulPTE` . . . 52
6.6 More Exotic Solutions - TLB Modifications . . . 53

7 Conclusion **54**

Bibliography **55**

Appendices **58**
A Kernel Functions . . . 59
A.1 `migrate_pages`[14] . . . 59
A.2 `unmap_and_move`[16] . . . 59
A.3 `mtier_unmap_and_move` . . . 59
B Algorithms . . . 61
B.1 Fisher-Yates Algorithm . . . 61

Vita **62**

List of Tables

4.1 Comparison of Benchmarks Used 29
4.2 Matrix Multiply Solve Times 30
4.3 Matrix Multiply Memory Bandwidth 30
4.4 Average STREAM Bandwidths in MB/s for Local and Remote Execution . . 31
4.5 STREAM Calculations Before and After Modification 32
4.6 Effect of Node Binding on Process Performance 34

5.1 TLB Shootdown Time Range 40
5.2 Comparison of Migration Types 42

List of Figures

3.1 The structure that maintains page status within the module 14
3.2 High-level overview of module behavior. 16
3.3 Bookkeeping iteration behavior. A failure at any stage results in a `break` and will be re-tried during the next iteration. 18
3.4 Standard iteration behavior. Any failure results in a `break` and will be re-tried during the next iteration. 20

5.1 Raw number of shootdowns with a 98MB high-performance memory size using varying delays and six threads. 38
5.2 TLB Shootdowns per core and total number of IPI sends 39
5.3 TLB Shootdowns per core and total number of IPI sends 40
5.4 Ratio of pages moved to shootdowns per core by `mtier` variant 42
5.5 Completion times for different types of page movements 43
5.6 Effect of Epoch Length on Performance. Lower is better. 44
5.7 Effect of migration function completion times on number of epochs completed in 1000 seconds 45
5.8 Effect of high-performance memory size on performance. Lower is better. . . 46
5.9 Effect of number of threads on performance for an `mtier` configuration with a 50 ms epoch as 12.5% benchmark RSS fast memory size. Values normalized to percentage of local execution baseline for respective number of threads. . . 47

5.10 Effect of number of threads on performance for an `mtier` configuration with a 50 ms epoch as 12.5% benchmark RSS fast memory size. Values normalized to percentage of local execution baseline for respective number of threads. This graph shows raw performance values. 48

1 Simple pseudocode for the Fisher-Yates shuffle algorithm [5] 61

2 Modified version of Fisher-Yates used in `mtier`. 61

Chapter 1

Introduction

The age of heterogeneous memory systems is here. In the past, the vast majority of computers had a single type of random-access memory with uniform bandwidth, latency, and clock speed. Even in non-uniform memory access (NUMA) systems, where there is extra overhead associated with one CPU accessing memory that is physically plugged into another CPU's memory sockets, the memory itself operates at the same clock speed and has the same bandwidth and latencies when accessed by its respective local CPU. In emerging heterogeneous systems, however, there are different types of memory with different bandwidths and latencies, and the higher-performance the memory, the lower its capacity, typically. Modern operating systems simply do not have the facilities to manage this type of memory effectively. For example, in Intel Knight's Landing Xeon Phi systems, the high-performance, near-die MCDRAM can be exposed as a separate NUMA node of addressable memory or used as a large memory cache. Even if exposed as a NUMA node, the operating system does not manage the memory in any special fashion: it is up to the end user to ensure that the memory is appropriately employed. Clearly, a solution to effectively manage these new types of memory in software is needed to employ them to their full potentials.

There are two primary ways to handle memory management in which a specific type or address range of memory is targeted. The first is to place all allocations of a certain type, belonging to a certain process, in the high-performance memory. This has the benefit of being relatively easy to implement and low-overhead, but it does not handle this high-performance memory being filled to capacity well. When it memory becomes full, the allocator falls back

to placing allocations in the low-performance memory. If one process consumes all of the high-performance memory, no other processes will be able to use it. If memory that is rarely accessed somehow becomes allocated in high-performance memory, it will remain there until no longer in use, potentially preventing other memory allocations from benefitting from the high-performance memory. In other words, the memory manager is not good at effectively sharing low-capacity memories among multiple workloads.

The second method of managing heterogeneous memory involves using the low-performance memory as a sort of swap space. Pages can be moved from a slow memory tier to a fast memory tier, and then quickly moved out at a specific interval, when they are no longer needed, or when there is contention. This management is performed by the operating system or a driver and allows multiple processes and multiple allocations within processes to share the high-performance memory without any one source being able to consume an inordinate amount of it. This method has numerous hurdles to overcome. There are multiple sources of overhead that arise when copying memory and changing memory mappings at frequent intervals. While data migration overheads are effectively negligible when migrations are performed infrequently, this can quickly change as the number and frequency of page and memory locks, translation lookaside buffer (TLB) shootdowns, and data copies increase. A technique is needed to minimize these overheads any memory management technique hopes to achieve rapid migration of memory from one tier to another within a hybrid memory system.

This thesis introduces the `mtier` Linux kernel module to minimize some of these sources of overhead. `mtier` takes advantage of the fact that many data objects, following a brief period of writes immediately after allocation, are effectively read-only or read-mostly for the duration of their lifespan[3]. By leaving copies of all pages promoted to high-performance memory in low-performance memory as well, `mtier` is able to achieve rapid eviction from the high-performance tier without incurring the overhead cost of a second data copy. `mtier` also coalesces memory management operations in a way that allows the number of explicit TLB shootdowns to be minimized when performing mapping changes in either direction. `mtier` does not require a lot of oversight or configuration from the user and, because it looks at all memory eligible for migration across all processes as a single entity and is epoch-based

instead of first-come-first-served, does not allow one process or group of allocations within a process to monopolize high-performance memory.

In the experiments performed for this thesis, `mtier` is stable and is able to achieve performance improvements vs. remote-node execution for certain workloads, particularly memory-intensive workloads that have large resident set sizes and are multithreaded. `mtier` is capable of achieving large performance improvements when compared to a userspace-only attempt at implementing the same memory management scheme. In addition to improving overall process performance, `mtier` performs its copying and swapping routines more quickly than the default kernel functions, reduces the number of data copies that are performed, and minimizes TLB flushing when changing memory mappings.

Chapter 2

Background

As stated in Chapter 1, no modern operating system has the facilities to effectively, automatically manage memory in a heterogeneous memory system. There has been an assumption for years (which admittedly is grounded in the reality) that a computer will not have different *types* of RAM. Indeed, most computer systems only allow the installation of one type of memory and, if different speed or latency memory modules are installed the system will degrade the performance of the faster modules to match that of the slowest module installed. Even today, most systems do not support multiple types of memory. However, thanks to the introduction of near-die memory in the form of MCDRAM in certain Intel Xeon Phi systems[10], it is now possible to purchase and create a heterogeneous memory system. DIMM-based persistent memory, which is slower but denser than typical DRAM, is also on the horizon in the form of various types of bulk-resistance-based memories like 3D-CrossPoint[2] and reRAM. As these technologies become more prevalent and less expensive, effective and generic solutions to manage them are needed to ensure they are used to their full extents.

However, no modern operating systems have these effective and generic solutions. Anyone who has a heterogeneous memory system must manually tune the operating system or any applications that need to leverage the memory. Ideally, the operating system would treat the lowest-performance memory as a sort of backing store for higher performance memories and allow swapping of pages into and out of the high performance memory in much the same way magnetic and flash-based storage devices are currently used for swapping. There

are, however, significant sources of overhead that must be overcome in order to accomplish this in a way that preserves performance. On Linux, for example, the memory migration framework operates on the assumption that any migrations are performed on read-write pages. This results in locking, process barriers, copy overhead, and TLB shootdowns that cause the migration process to become a bottleneck if performed frequently.

In reality, most objects in memory are in practice read-only or read-mostly following a brief period of writes after its initial allocation[3]. This provides a starting point for research on reducing the overhead of frequent migrations by taking advantage of the fact that most old objects in a process' memory space (and therefore most of the process' memory space once steady-state is reached) will probably not be changed again and that the machine's lower-performance memory capacity will be substantially higher than its high-performance memory capacity. If a copy of a memory page is left in low-performance memory when its high-performance counterpart is being used, there is no need to incur page copy overhead when the mapping needs to be switched from high-performance to low-performance memory. Additionally, there is no need to flush the TLB explicitly when moving memory from low-performance memory to high-performance memory. Since the slow mapping is still valid, any "stale" TLB entries can be used until a TLB refresh happens organically e.g. via a context switch. This saves the overhead of having to explictly flush all or part of the TLB when page mappings are moved into high-performance memory.

2.1 Production Systems

As previously stated, no modern production operating systems have the ability to manage a heterogenerous memory system in the manner described by this paper. While swapping does exist, it exists solely as a method of reducing memory consumption by moving pages to and from disk (magnetic hard drive or solid-state drive). Linux, for instance, has a number of system calls that can move memoryfrom one NUMA node to another, but the migration tools are geared more towards load balancing. For example, the `numactl`[13] command allows a NUMA policy to be set when a process is loaded that attempts to guide allocations and processor affinity of the processs while it runs. It has extremely limited functionality for

working with processes that are already running. The `migrate_pages` system call[14][1], for instance, allows the entire address space of a process to be moved to a different NUMA node, but does not allow pages to be selectively moved. The `mbind` system call[8], on the other hand, allows single pages to be moved, but puts a burden on the application developer to ensure allocations are properly page-aligned. For it to support rapid and targeted migrations of individual pages, it has to be run in a loop in which it is called potentially thousands of times, preferably in its own thread to keep it from interrupting process execution. Finally, overuse of `mbind` on discontiguous pages can result in virtual memory areas (VMAs) being split so many times that the process exceeds its kernel-enforced limit; the kernel must be properly configured to increase this limit or `mbind` calls will quickly start to fail.

Some near-die memory systems are currently available. For example, the Intel Xeon Phi Knight's Landing processors are available with up to 16 GB of "high-performance, low-capacity" MCDRAM[10]. Out of the box, these systems allow this memory to be configured as extra memory in the system's address space (exposed as a separate NUMA node), as a large cache, or split in various ratios between extra general-purpose memory and cache. Any novel memory management techniques, like those outlined in this thesis, must be handled by software. In other words, in its default configuration and without any manual memory binding on the part of the user, the underlying operating system does not treat this memory any differently than it treats generic DRAM, despite the fact that there are substantial differences between the two types of memory.

2.1.1 Experimental Systems

A number of experimental frameworks exist to address issues with targeted memory allocation (of which heterogeneous memory management can be seen as a sort of subset) or heterogeneous memory management explicitly.

`X-Mem`[7] moves process memory between tiers (or rather, performs allocations on the most beneficial type of memory to begin with), but does not have online support and requires an

[1]The `migrate_pages` system call is easy to use and works on running processes, but does not allow any sort of targeted migration. It will attempt to move every page in a process' address space, and will only not move a page if some policy prevents it or if the initial attempt to migrate fails

expensive binary profiling step to function correctly. It is also not designed for frequent memory movement; `X-Mem` uses the default Linux system migration framework to handle memory movement when needed. While it is designed with heterogeneous systems in mind (particularly systems containing nonvolatile memory), it does not truly address the costs of data movement or analyze sources of overhead.

The `mcolor` framework[12] is another academic work that allows targeted memory placement. In `mcolor`, memory is divided into regions based on physical address ranges. These ranges are user-defined and can be based on of any arbitrary division of physical addresses, even discontiguous divisions, for example DIMMs or even DIMM ranks. Regions can then be assigned "colors," which are in turn assigned allocation policies. For example, one range of physical addresses can be assigned a color that is reserved for allocation of frequently-accessed objects while the rest of the system memory is assigned a color that is reserved for infrequently-accessed objects. This can result in power consumption reduction on the machine by allowing most of its memory to spend more time in a low-power state. `mcolor`, however, requires a special kernel and does not allow for rapid movement of pages or automated memory management; policies must be configured by the system administrator.

`Carrefour`[6] allows pages within a process to be duplicated so each socket in a NUMA system has its own local copy of process memory. However, `Carrefour` is geared more towards memory placement to reduce memory system congestion and accomplishes this by creating a separate page table hierarchy for each CPU (e.g. a separate `cr3` register value for each processor on x86-64). This can result in large increases in page table size as the duplication of a single page requires, at the very least, a new page table and could require new page directory, page directory pointer, and PML4 pages[2][11]. In a worst-case scenario (requiring as few as 1/512 of the process' pages, depending on distribution of the duplicated memory), the paging structure size overhead could be increased by 100% *per core*.

[2]In an x86-64 system using 4 KB pages, the PML4 contains 512 pointers to page directory pointers, which contain 512 pointers to page directories, which contain 512 pointers to page tables, which each contain 512 page frame numbers that map a virtual address to a physical address.

2.2 What's Missing

None of the experimental or production systems mentioned truly address the costs of data migration overhead. If any sort of low-overhead, rapid, targeted migration is to be achieved with byte-addressible memory in hybrid architectures, these sources of overhead must be addressed and minimized if possible. This will reduce the burden on users, system administrators, and application developers who use or write programs that will run on hybrid memory systems and, more importantly, allow more equitable sharing of less-dense memories among processes or allocations within a process when compared to current systems that can succumb to greed.

The primary questions that need to be answered revolve around the overhead of migration operations. Two major sources of overhead were identified: TLB shootdowns and page copies. The time required to perform the shootdown itself (between a couple hundred nanoseconds[4] and a substantial 3µs[17]) cannot be reduced, but reducing the number of shootdowns that occur is a possibility. However, both the time required to perform a page copy and the raw number of page copies have potential for reduction. Once these sources of overhead were addressed, questions related to equitable sharing of high-performance memory and choosing when and what to migrate were answered.

Chapter 3

The `mtier` Framework

To test both the feasibility and effectiveness of an automated hybrid memory management system, a framework called `mtier` was created. `mtier` consists of both a number of modifications to the Linux kernel (version 4.4.17), a kernel module that can be loaded and unloaded at will by the user, and a small helper program. The module contains the bulk of the framework. The helper program is needed to register eligible processes with the `mtier` framework. The kernel changes can easily be removed; they exist to maintain compatibility with some previous work but could trivially be removed from the kernel and made a part of the module itself.

3.1 Principles of Operation

`mtier` is an epoch-based framework. It operates on each eligible process at an interval specified when the module is loaded. This interval can be as short as roughly 50 milliseconds; shorter intervals can reduce timing precision. It attempts to treat the memory for all eligible processes as a single block to avoid preferentially managing memory for any single process. To simulate a hybrid architecure, two NUMA nodes are used. One is specified as the fast node and has process execution bound to it, the other is specified as the slow node. When loaded, the module reserves a user-specified amount of memory on the simulated fast node to represent the less-dense high-performance memory.

The key principles behind `mtier`'s operation are **page shadowing** and **TLB shootdown batching**, and both of these principles rely on the fact that most data is read-only following initialization[3].

Page shadowing, the initial goal of the research that led to `mtier`, involves leaving a copy of a migrated page (the *shadow page*) at its original location and storing the original page table information when the page is promoted to high-performance memory. If the data is read-only, then if the high-performance page is evicted, it can very quickly be reverted to its slow-memory mapping by switching the PTE back. Additionally, because the slow page is never released, no explicit TLB flush is required when a page is promoted; the process can continue to access the shadow page until a TLB flush occurs organically (e.g. via a context switch).

TLB shootdown batching arose from a concern present during the development of page shadowing. While shadow pages remove the need to perform a TLB shootdown on promotion, they do not allow a similar guarantee when a page is *evicted* from the fast memory. Since an eviction should not occur unless another page on the slow tier is being promoted into a fast page frame that is already in use, failure to flush the TLB will result in processes accessing invalid data. However, TLB shootdowns are high-overhead[17]. To minimize the impact of these required shootdowns, `mtier` is designed to perform all its evictions at once during each epoch and, only once all evictions are complete, perform a complete TLB flush *before* any new pages are promoted to the fast tier.

3.2 Kernel Modifications

One of the design goals of `mtier` was to keep as much of its functionality kernel-independent (i.e. in the module) as possible. Despite this, some modifications to the kernel were necessary. The first, and simplest, modification are three changes to the process control block (`struct task_struct`. A boolean value is added that specifies whether or not the process is eligible for `mtier` operations. A second boolean value is created that is set to `true` when the process is trying to exit. Finally, a lock that is held when the process is exiting is added to prevent

the module from trying to move any process memory to the high-performance memory and to allow the module to perform its cleanup steps.

The next, and more substantial changes, to the kernel involve the memory manager itself. A decision was made early in the development of `mtier` to operate on pages instead of data structures or primitive values, so the existing kernel non-uniform memory access (NUMA) migration framework was used as a starting point for development of the framework. methods First, the default NUMA migration core functions were duplicated and exported to allow them to be accessed from modules instead of just from within the kernel proper. Then, to maximize performance, all unnecessary code paths were removed: since `mtier` only operates on heap memory, the only necessary functionality is that which focuses on anonymous pages. Next, since the entire benefit of shadow pages and TLB shootdown batching is lost if pages can become dirty, the new `mtier` functions were modified to only consider read-only pages as eligible for operations. Finally, the function that assesses eligibility of pages is redesigned to pass the list of eligible pages it generates back to the module instead of using it directly within the kernel.

These modifications differentiate the `mtier` kernel operations from standard NUMA operations in two key ways. First, explicit NUMA migration queues the entire memory address space of the process (excepting some special cases like file-backed pages) for migration, as opposed to `mtier` which queues only a portion of the process address space for duplication. Second, the default NUMA migration functions do not submit their generated page list for further analysis and manipulation whereas `mtier` maintains the page list for its next steps.

It is important to note that these kernel changes do not have to be in the kernel. The long-term goal for `mtier` is to make it a standalone module that does not require any additional software (besides the helper program). These functions have been left in the kernel to ease certain instrumentation and for compatibility reasons, but removing them will be fairly simple.

3.3 The Helper Program

The helper program is the simplest component of `mtier`. When invoking an executable that should be eligible for page duplication, the user must invoke it with this program. It ensures that the eligibility value in the process' process control block is set and spawns an instance of the appropriate executable using `fork()` and `exec()`.

3.4 The Module

The `mtier` module ("`mod_mtier`") is designed to perform the bulk of the work related to hybrid memory management. When the module is first loaded, it reserves the entire high-performance memory area for itself. It also creates an array of `tier_struct`s to manage the high-performance memory and handle any duplication and swapping that becomes necessary.

The module also maintains a number of linked lists: one to keep track of pages eligible for duplication, one to track free `tier_struct`s, and one to track used `tier_struct`s. These lists are crucial for both performance and bookkeeping.

3.4.1 The `tier_struct`

The `struct tier_struct` is crucial for `mtier` to function correctly and should be explained in detail. One instance of this structure exists for each page of high-performance memory. It contains:

- The process ID of the process that is using or that last used the high-performance page referenced by this instance of `struct tier_struct`. A value of zero indicates that the high-performance memory has not been used since allocated.
- Boolean values to show whether or not the high-performance memory referenced by the structure is in use and valid. A false value for either of these variables allows the module to give the memory to another page on its next iteration.

- The page frame number and virtual (kernel linear, not process virtual[1] address of the high-performance memory referenced by this structure. These values will always be populated unless the `mtier_massive` variant is being used.
- The page frame number and virtual address of the low-performance memory referenced by this structure. These values do not have to be populated and may contain invalid values if the structure is no longer valid or the high-performance memory has fallen out of use.
- The userspace virtual address of the page within its process' address space.
- The VALUES of the page table entries for the slow page and the fast page.
- A pointer to the page table entry for the page within the process' page table hierarchy.
- A pointer to the page table lock for the process' page table.
- References to the `page_structs` for both the high-performance and low-performance memory referenced by this structure. These are used for reconstruction of state flags and locating virtual memory areas. The reference to the high-performance `page_struct` will always be populated and will not change; the reference to the low-performance `page_struct` may not be populated, may change as management operations occur, and may contain invalid values and should therefore not be relied upon unless the `tier_struct` is valid.
- A `list_head` structure that allows the `tier_struct` to move between the free and used linked lists.
- An `hl_node` structure that allows the `tier_struct` to be added to a hash table based on its slow page's kernel linear address.

See figure 3.1 for a detailed, in-order, typed definition of the `tier_struct`.

[1]Each page owned by a userspace process has two virtual addresses: the kernel linear address and the process virtual address. The process virtual address is useless for managing memory since the kernel does not directly use these addresses for internal memory management and has no fast way of reconstructing any other page address information from these addresses.

```
struct tier_struct {
    pid_t owner;
    int fast_in_use;
    int fast_valid;
    unsigned long fast_pfn;
    unsigned long fast_vaddr;
    unsigned long slow_pfn;
    unsigned long slow_vaddr;
    unsigned long usr_vaddr;
    pte_t fast_pte;
    pte_t slow_pte;
    pte_t *pte;
    spinlock_t *ptl;
    struct page *fast_page;
    struct page *slow_page;
    struct hlist_node hl_node;
    struct list_head list;
};
```

Figure 3.1: The structure that maintains page status within the module

3.4.2 The Linked Lists and the Hash Table

Originally, `mtier` simply maintained an array of `tier_structs`. However, as larger and larger high-performance memories were used, the lookup time for finding a free high-performance page became prohibitively high. The use of linked lists results in a slightly higher memory footprint for the module but allows instantaneous lookup of a free `tier_struct`. This was considered a reasonable tradeoff as performance is the primary goal of the module. The page list, on the other hand, is a necessity due to the way the kernel identifies pages that are eligible for duplication.

The hash table is a fairly memory-intensive data structure but is necessary for fast lookups of used `tier_structs` by virtual address. When a standard iteration is performed, some pages that are in fast memory are going to remain there. To mitigate the performance overhead associated with evicting these pages and then putting them right back, pages are added to the hash table when they are moved to fast memory. The hash key is the linear address of the slow page with the 12-bit offset shifted out. Under the hood, the hash table

is chained, although it is large enough that collisions should be rare unless the fast memory size is extremely large.

3.4.3 Operation

`mtier` takes four parameters when the module is loaded: the size of the high-performance memory in megabytes `s`, the epoch length in milliseconds `e`, the fraction of the high-performance memory to fill each iteration `f`, and the frequency of bookkeeping iterations `b`.

The first thing the module does is reserve $s/page\_size$[2] pages on the simulated high-performance memory. A `tier_struct` is created for each of these pages and added to the free list. A thread is then spawned that will run until the module is explicitly unloaded.

The thread performs the actual memory management. It sleeps for `e` milliseconds and then awakens to perform one of three iteration types (see sections 3.4.4, 3.4.5, and 3.4.6 for iteration details). How it behaves is determined by whether or not its current iteration is a bookkeeping iteration. If $iteration\%b == 0$, the thread performs a **bookkeeping iteration**. Otherwise, it performs a **standard iteration**. If no eligible processes are found, the iteration is dubbed a **dummy iteration**. Figure 3.2 contains a flowchart showing startup and general runtime behavior of the module.

It should be noted that `mtier` is designed so that if a process is loaded using the helper program without the module loaded it will just run normally. Additionally, if the module is unloaded while a tiering-eligible process is running, the process should be cleanly restored to its default state before the module is removed; however, this is not recommended with the module in its current state.

3.4.4 Bookkeeping Iterations

Bookkeeping iterations are performed by the module every `b` iterations (so, at most, every $b * s$ milliseconds. During a bookkeeping iteration, every `tier_struct` in the used list is examined. For each of these structures, the fast page's mapping is copied to the slow page

[2]Typically 4 KB, but variable and even user-configurable depending on architecture and operating system support.

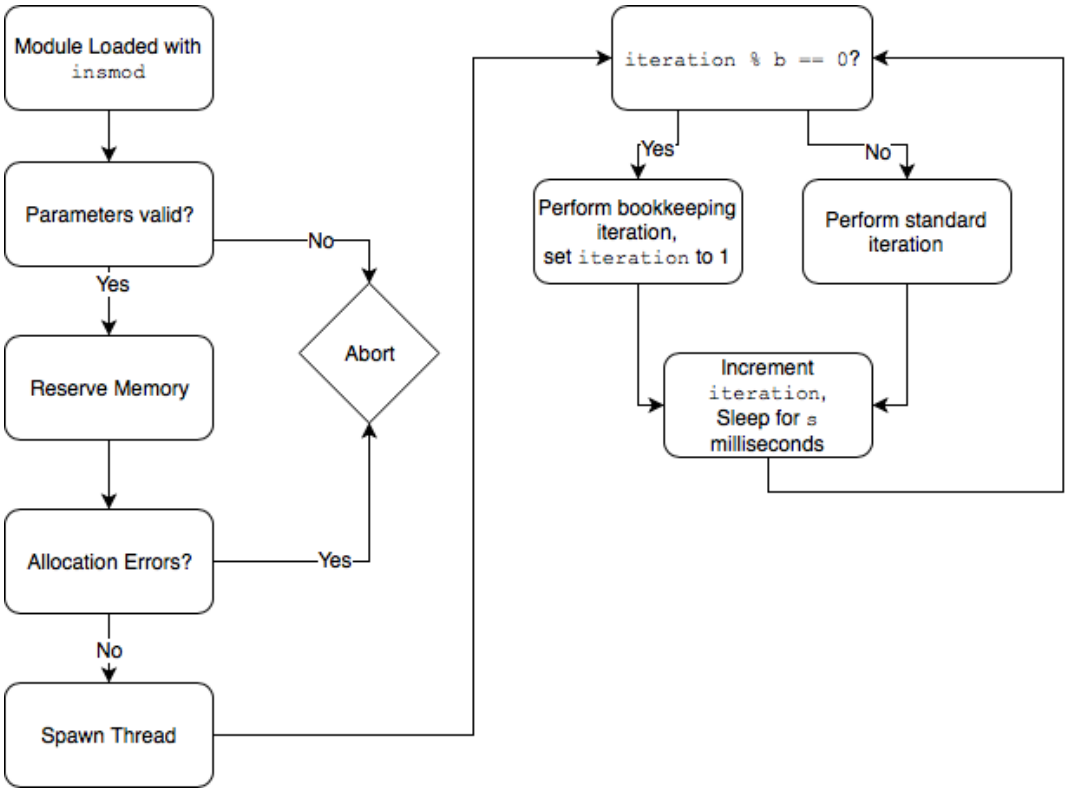

Figure 3.2: High-level overview of module behavior.

and a page table entry (PTE) swap is performed to switch the page actually used by the process to the low-performance memory. The `tier_struct` is then moved back to the free list. This serves to evict the entire high-performance memory, providing a clean slate on which the next standard iteration can operate. The page table entry address and values for both the fast and slow mappings are stored in each used `tier_struct`. This ensures that no page table walks or other time-intensive lookups need to be executed in order to perform the swap. This process can be performed without taking any page or page table locks, which further helps provide seamless performance. The actual swap is essentially a single line of code that replaces the value stored at the page table entry address with the value stored in the slow page table entry.

The page list is then regenerated using a call into the kernel. This ensures that no invalid pages are held in the page list and that new duplication-eligible processes have their pages added for duplication consideration. Therefore, when a new tiering-eligible process is loaded, there is are roughly $b * s$ milliseconds at most before tiering operations are performed on its memory; for processes with long runtimes, this is not a substantial-enough delay to have a long-term impact. See figure 3.3 for a flowchart outlining this behavior.

3.4.5 Standard Iterations

During a standard iteration, the module first allocates an array of `page_struct` pointers with a number of elements equal to the size of the page list. It then uses the kernel's non-blocking pseudorandom number generation capability to shuffle these pointers so that each element in the array points to a mostly-random (depending on available kernel entropy) element of the page list. The Fisher-Yates algorithm (see figure B.1) [5] is used to perform this shuffle. Attempting to ensure true randomness can cause long-term blocking if the kernel's entropy pool is depleted, so a non-blocking function that falls back on pseudorandom number generation if insufficient entropy is available is used instead of a blocking, truly random (i.e. cryptographically-secure) number generator. Considering that the module does require the level of security provided by the blocking function, this is an acceptable tradeoff to keep performance within acceptable limits. The first n pages pointed to by this shuffled array (where $n = MIN(maximum\_migration\_size, pagelist\_size)$ are now eligible for duplication.

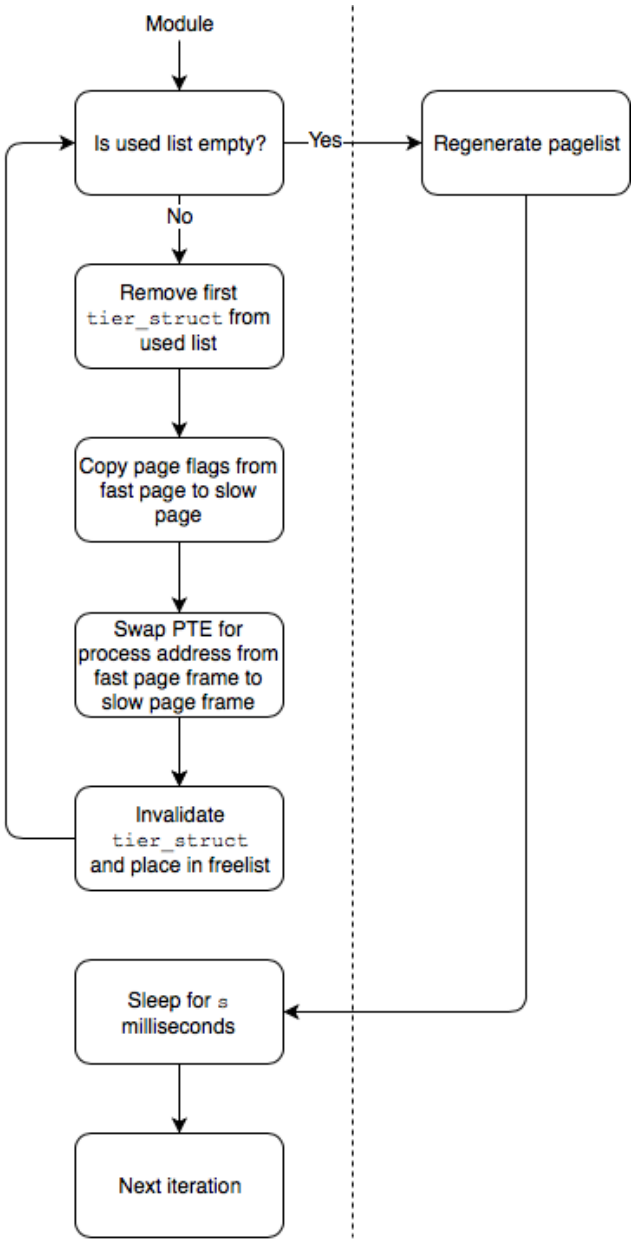

Figure 3.3: Bookkeeping iteration behavior. A failure at any stage results in a `break` and will be re-tried during the next iteration.

Once the pagelist is shuffled, each entry in the used list has its `tier_struct` invalidated but no swap backs are performed. Then, each entry eligible for duplication is searched for within the hash table to see if it is already present in high-performance memory. If the page is found, the corresponding pointer in the shuffled array is set to NULL to signify that it should not be duplicated and its `tier_struct` is re-validated. Then, each entry in the used list that still has an invalid `tier_struct` is fast-swapped back to the slow tier and the `tier_structs` are moved from the used list to the free list.

Next, a linear scan of the shuffled pagelist pointers is performed. For each non-NULL pointer encountered, a free `tier_struct` is obtained from the free list and the corresponding page is copied to the fast tier and its mapping and page table entries are swapped from the slow page contained within the pagelist to the fast page contained within the `tier_struct`. A walk of the process' virtual memory areas must be performed to reconstruct its userspace virtual address and find the appropriate page table to modify, so this process is substantially slower than the fast swaps performed during evictions and bookkeeping iterations. Because of this, all information gained during this walk is stored in the `tier_struct` to allow future manipulations to be performed on still-valid entries without performing the memory area walks a second time (see figure 3.1 to see which information is stored).

Finally, the selected `tier_struct` is moved from the free list to the used list and is added to the hash table keyed on the slow page's kernel linear address. When all these steps have been completed successfully, the next page in the shuffled pagelist is examined and moved, if necessary. When the selected number of pages have been moved, the module thread goes back to sleep for e milliseconds.

If, at any point during this process, the free list or page list become empty before the correct number of pages have been examined, the process aborts and the next iteration is designated a bookkeeping iteration. See figure 3.4 for a simplified flowchart of this complex iteration behavior.

3.4.6 Dummy Iterations

A dummy iteration occurs when no duplication-eligible processes are found. It could start life as either a bookkeeping iteration or standard iteration. When this state occurs, the

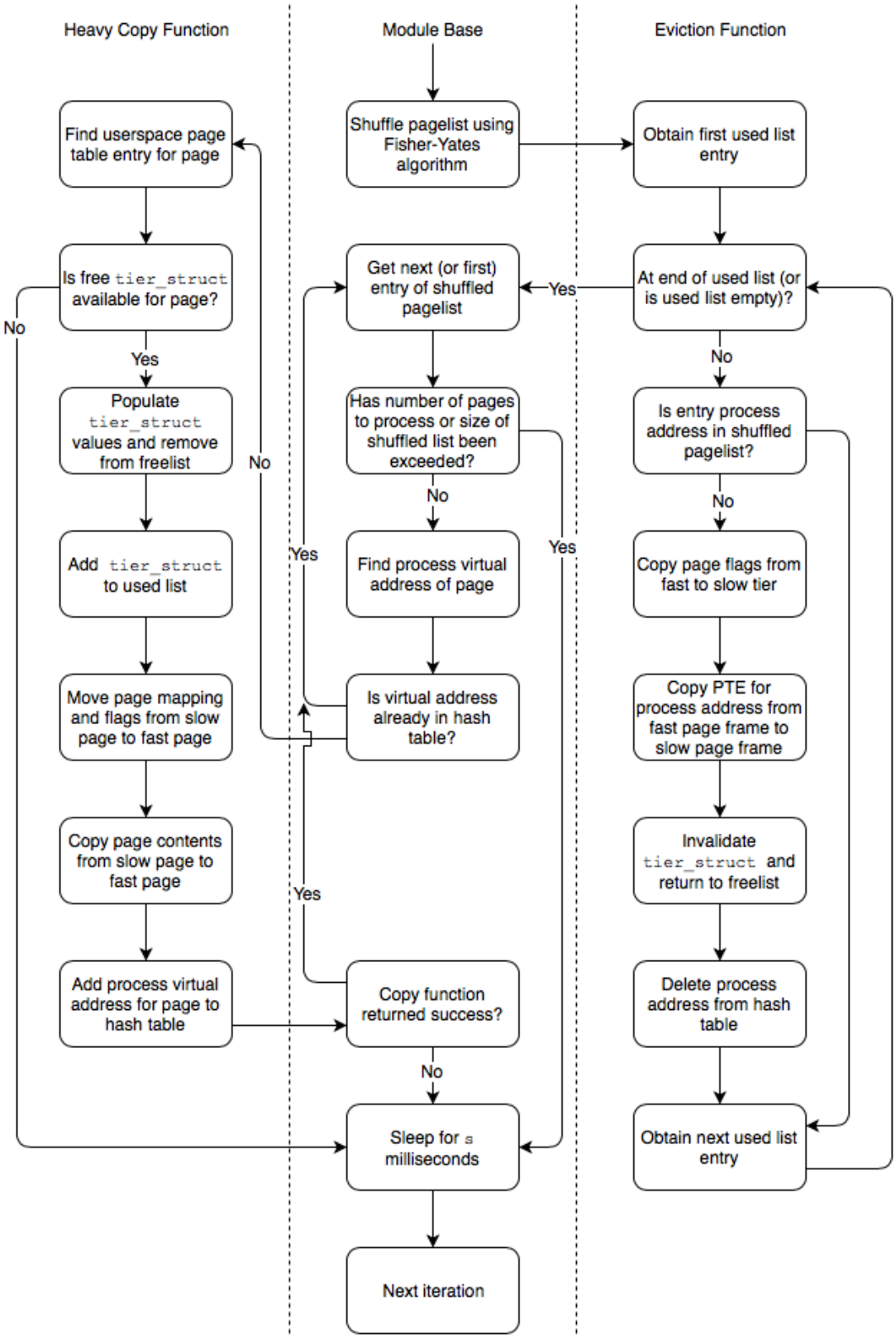

Figure 3.4: Standard iteration behavior. Any failure results in a `break` and will be re-tried during the next iteration.

module enters a state that will ensure the next iteration will be a bookeeping iteration. This guarantees that a page list will be populated as soon as eligible processes are found.

Because the module is epoch-based, the vast majority of iterations may be bookkeeping iterations if no duplication-eligible processes are ever invoked. Unfortunately, this consumes some amount of system compute resources whether or not any work is actually performed. This is considered a fair and necessary tradeoff.

When any iteration of either of the three above types is complete, the thread will sleep for `s` milliseconds and then wake up to perform another iteration. Note that this means the gap between the start of two iterations could be much longer than `s` since the time spent performing work is not counted as part of the gap between iterations. Because the duplication-related work can take longer than `s` milliseconds, attempting to guarantee an exact iteration time of `s` proved to be futile.

3.5 Eviction Modes

`mtier` has three eviction modes used to handle different cases. These eviction functions are used to swap pages that are in use on fast memory back to their slow memory counterparts. The actual process to swap the page table entries and mappings are the same for each eviction mode; fast page eligibility for eviction is the only thing that changes between eviction modes.

3.5.1 `evict_unused`

`evict_unused` is employed during standard iterations to clear pages that are no longer needed from the fast tier. Each `tier_struct` in the used list that is marked as invalid is swapped from the fast tier to the slow tier then moved from the used list to the free list.

3.5.2 `evict_pid`

`evict_pid` is used when a process is exiting to return it to its default state prior to returning its memory to the kernel. Each `tier_struct` on the used list is examined and, if it is valid and its `owner` field matches the `pid` of the process selected for eviction, its mapping is

swapped from fast to slow memory. The `tier_struct` is then moved from the used list to the free list.

3.5.3 `evict_all`

This eviction mode is employed when the module exits. If any fast-tier pages are in use, they are swapped from fast to slow memory and invalidated. This allows any running process to have its default state restored before the manager stops running. Since this mode is only employed when the module exits, the fast pages are also freed when the swap back is performed to allow the memory claimed by the module to be returned to the kernel.

3.6 Handling Edge Cases

The module's basic operation (standard and bookkeeping iterations) does not allow for process exitor early module termination. Failure to explicitly handle these cases results in a barrage of errors and possibly a system crash when one of them occurs.

3.6.1 Process Exit

When a process exits, the kernel function `do_exit()` is called and will, among other things, free the process' memory. If the process is using any fast-tier memory when this happens, the kernel will generate a "page charge error" for each affected page, since the fast pages are not reclaimable and technically still belong to the module. Additionally, if a process exits while a tiering operation is underway, the sudden presence of newly-invalidated pages in the pagelist can crash the system.

To get around this, an "exit" semaphore has been added to the process control block structure. When a standard or bookkeeping iteration is underway, the module holds the semaphore for the task it is currently manipulating. When a process exits, the `do_exit` function instead holds the semaphore. This prevents the kernel from reclaiming the process' memory when the module is managing it and prevents the module from trying to manage

memory for processes that are exiting. If `do_exit` cannot obtain the semaphore, it sleeps and tries again in the future.

When `do_exit` obtains the semaphore, it then waits on the semaphore again. When the module begins its next iteration, it detects that the semaphore is held, which can only signal that the process is attempting to exit. It then initiates a per-PID eviction of all fast memory used by the process (see section 3.5.2) and releases the sempahore, allowing `do_exit` to continue its work.

This eviction step is independent of the usual module iterations. When a process is exiting, its fast memory is evicted prior to the execution of whatever type of iteration the module is currently scheduled to perform.

3.6.2 Module Exit

When the module exits, an all-pages eviction (see section 3.5.3) is performed to clear the entire high-performance memory. This allows the module to free this memory without destabilizing any running task or leaving unreclaimable pages present in either fast or slow memory. This happens instantaneously when the `rmmod` command is used to remove the module from the kernel.

Despite allowing for this, it is not safe to remove the module when a tiering-eligible process is running. Additional work is being performed to make this operation safe, but for the time being all processes should either be terminated or allowed to exit normally before removing the module. This is acceptable behavior while the module is still in an experimental state.

3.7 Module Experimental Configurations

3.7.1 `mtier_heavy`

The `mtier_heavy` configuration was created to attempt to quantify the performance gain from using the faster migration and swapping code present in the base `mtier` module. It has no lightweight swaps used for page evictions and does not use `mtier`'s copying and duplication

code, although its basic operation is unchanged. `mtier_heavy` still uses pre-allocated pages on the fast tier, but calls into the kernel to perform all swaps using a modified version of `unmap_and_move` called `mtier_unmap_and_move` (see section A.3).

During standard iterations, `mtier_heavy` uses this modified function to perform all copying and page table manipulations. In addition, during evictions of pages from the fast tier (e.g. during bookkeeping iterations), it uses the same function instead of attempting to perform a higher-performance swap. This allows performance using mostly-standard kernel faculties to be measured and compared to the base `mtier` module.

`mtier_heavy` uses the same methods as the base `mtier` module for tracking used pages, generating the pagelist, and determining which pages are going to be moved from the fast tier to the slow tier. This similarity in operation allows just the migration overhead differences to be measured.

3.7.2 `mtier_massive`

`mtier_massive` is a modification of `mtier_heavy` that does not use pre-allocated fast-tier pages. It is designed to measure any additional overhead that may exist due to the default kernel behavior of allocating pages only when needed. Because of this, when migrations are performed, it allocates a new page on whichever node is the destination and frees the old page when the migration is complete. While it still uses `mtier_unmap_and_move` to perform the targeted migrations, this allocation provides an almost exact duplicate of the kernel's default migration overhead, although the migration is still targeted per-page as opposed to being for the whole process.

3.7.3 `stream_madvise` and `stream_mbind`

These configurations explore the possibility of a userspace-only approach to memory movement. The `mbind` variant uses the `mbind()` system call to bind certain process virtual addresses to certain NUMA nodes, while the `madvise` variant attempts to use the `madvise()` system call to free pages and force page faults during the movement process. The

randomization algorithm used is the same as the above module variants, but some additional considerations had to be made to allow these two variants to work.

First, any memory eligible to be moved has to be page-aligned, which isn't something that's guaranteed using the standard system `malloc()` function. `mmap()` is used instead to force alignment. Second, the entire area of memory eligible to be copied must be contiguous. To ensure this, it is generated beforehand, copied to a ramdisk on the test system, and then `mmap`ed to a single array. Third, as the name implies, this requires modification of the benchmark itself and not the use of a generic module. To accomplish this, the STREAM benchmark was modified to have a separate thread that would activate at set intervals and perform the memory management.

Despite these efforts, there were significant difficulties getting the `madvise` variant to consistently work well. It seems to be highly dependent on system and ramdisk configuration, and therefore would not always behave in the desired the manner. The `mbind` variant was created later and does work as intended.

3.8 Module Development Configurations

During development of the `mtier` framework, two incremental frameworks were developed to test certain principles before commiting to the more-complex and time-consuming full framework. Neither of these modules has yielded results that are presented in this paper (although the `mtier_heavy` and `mtier_massive` experimental configurations are similar to the second development configuration), but they are important for historical reasons.

3.8.1 `mtier_dev_full`

This framework was used to ensure that epoch-based migration could be used without destabilizing the system or causing issues with the running process. It used the kernel's standard `migrate_pages` (see section A.1 for an explanation of this function) function to move the entire address space of the process between two NUMA nodes at set intervals. Performance was poor, but it showed that frequent migrations, in and of themselves, did not cause any problems.

3.8.2 `mtier_dev_partial`

Following development of the previous test configuration, a question remained about whether or not random pages could be migrated without causing problems. In this configuration, the module generated the list of pages eligible for migration, randomly selected which pages would be moved, and then used a modified version of the kernel's `unmap_and_move` function (see section A.2 for an explanation of this function) to move only the selected pages. Performance was not optimal, but this configuration showed that selective migration was possible. This configuration did not duplicate pages as the final version of `mtier` does; each time a migration is performed a new page must be allocated in the destination memory region and the original page is freed and returned to the operating system.

3.9 Deficiencies

There are a number of deficiencies with the module-driven, epoch-based approach that could be overcome with future work.

3.9.1 Multi-Process Behavior

For experimental purposes, the module only manages one tiering-eligible process at a time. Some issues have been identified with the module in its current state that would prevent it from effectively managing multiple processes. These issues mostly revolve around bookkeeping and do not involve the basic principles of the module's operation. While the module is in an experimental state they have not been resolved, but will be if the module is developed into a production-ready system.

3.9.2 Early Module Exit

If the module exits while a tiering-eligible process is running, there can be questionable behavior as the processes exit. As with the previous issue, this involves bookkeeping and not the underlying operation of the module itself. As with multi-process behavior, this exit behavior will need to be resolved before the module can be released for production systems.

3.9.3 Mixed Kernel-Module Approach

Ideally, the entire bulk of the `mtier` framework would exist inside a module and would not require a special kernel (short of a supported version, of course) to function. Due to the complexity of the kernel's memory manager, this is a daunting task that would not be easy to complete. However, some functionality remaining within the kernel proper should be acceptable for long-term study and even production use.

Chapter 4

Experiments

4.1 Hardware

All experiments detailed in this paper were run on an x86-64 server. The machine consists of two Intel Xeon E5-2620 processors running at up to 2.1 GHz. Each CPU has six cores with hyperthreading for a maximum of 12 threads per CPU and 24 for the overall machine; however, due to a concern that HyperThreading was masking some system overhead during early experiments, it was disabled and all experiments were re-run. This results in a total of 12 execution threads (six per socket). Each core has a 32 KB L1 data cache, a 32 KB L1 instruction cache, and a 256 KB L2 cache. Each CPU shares a 15 MB L3 cache among its six cores. The system has 64 GB DDR3 in the form of eight evenly-size DIMMs running at 1600 MHz. The memory is evenly split between the two sockets, so each NUMA node has 32 GB RAM. An Intel QuickPath Interconnect running at 25.6 GB/s exists between the two sockets for inter-socket memory accesses [9].

4.2 Benchmarks Used

Two basic experiments were used to test both the `mtier` framework and the overall concept of automatic, software-based hybrid memory management. The first is a double-precision matrix multiplication program[1] that is used mainly to test functionality due to its inability

[1] Solving $[a] * [b] = [c]$

to consume large amounts of memory bandwidth. The second is a modified version of the STREAM benchmark that tests performance and memory utilization.

Both benchmarks do not work in their typical fashion. Since `mtier` operates on read-only memory and part of its purpose is to ensure that memory duplication is, in fact, possible, no results are written to their destination arrays. To reduce memory writes and focus as much as possible on reads, no writes are performed on the destination matrix for either benchmark. Results are stored in single variables and essentially discarded so the vast majority of observed memory traffic comes from reads. This decision was made after noticing that large cross-NUMA-node memory writes could artificially inflate the "read" bandwidth on one or both nodes. Each benchmark has also been modified to call `mprotect`[2] on each array from which it reads to make the array read-only and therefore eligible for duplication.

Table 4.1: Comparison of Benchmarks Used

Benchmark	Time	Threads	Memory Usage	Memory Bandwidth
Matrix Multiply	Variable	Single	Variable	Low
STREAM	Short (enforced)	Multiple	High	Low - Very High

4.2.1 Matrix Multiply

The matrix multiplication benchmark is a relatively fast benchmark (when small matrix sizes are specified) used for testing general concepts and technical functionality of the `mtier` framework. It accepts a number of command line arguments for matrix size, a random seed, the number of times it should be run, and whether or not `mprotect` is used to mark the source matrices as read-only. It is a double-precision multiplication, so each matrix element is 64 bits (eight bytes) in size. The total size of the source matrices can be calculated by $8 * matrix\_size * matrix\_size * 2$. Because of the way `mprotect` works, each source matrix's size must be a multiple of the system's page size (4 KB on the test machine used for this paper). This can be easily guaranteed by selecting a matrix size that is both a power of two and greater than or equal to 1024 x 1024.

[2]The `mprotect` system call allows pages to be made readable, writeable, and/or executable. For these benchmarks, the `PROT_READ` flag is provided to remove all other access permissions from the memory.

The matrix multiply benchmark is, by design, single-threaded, although it could be easily modified to be multi-threaded. Because of this it is not a large driver of memory bandwidth, although with the right arguments it can have a substantial resident set size (see Table 4.2). However, as matrix size increases the time needed to solve a single matrix increases non-linearly, making this an impractical benchmark for tests where large amounts of memory need to be consumed. For a 16384x16384 matrix, the solve time was so long that attempts to find it were aborted. Table 4.3 shows the relatively low bandwidth consumption of the matrix multiplication benchmark. Given this low bandwidth consumption compared to STREAM (see Section 4.2.2), the matrix multiplication benchmark demonstrated itself as unsuitable for anything other than routine testing of the `mtier` framework.

Table 4.2: Matrix Multiply Solve Times

Matrix Size	Source Size (MB)	Source Size (Pages)	Solve Time (Seconds)
1024 x 1024	16	4 096	10.203
2048 x 2048	64	16 384	257.107
16384 x 16384	4 096	1 048 576	Prohibitive

Demonstration of non-linear increase in matrix solution time as matrix dimensions increase.

Table 4.3: Matrix Multiply Memory Bandwidth

Matrix Size	Peak Bandwidth	Average Bandwidth
1024 x 1024	7.87 MB/s	6.85 MB/s
2048 x 2048	2062.75 MB/s	2047.53 MB/s
16384 x 16384	3915.04	2200.59 MB/s

Demonstration of low bandwidth consumption by the matrix multiplication benchmark.

4.2.2 STREAM

Due to the poor performance of the matrix multiplication benchmark, an alternative was needed to truly evaluate the performance of the `mtier` module. STREAM was selected. STREAM is a bandwidth measuring benchmark that is the "de facto industry standard[15]." Unlike the matrix multiplication, STREAM is multithreaded (courtesy of OpenMP), which allows it to substantially increase the amount of memory bandwidth it consumes while running. It is also not quite as easily-configurable as the matrix multiply: its array sizes must

be set at compile time. Stream performs a series of mathematical operations on two source matrices and stores them in a destination matrix. These operations include copying, addition, multiplication by a constant scalar, and addition of a result obtained via multiplication with a constant scalar. The data types STREAM uses are configurable; for this paper, all operations are double-precision. Table 4.4 shows how STREAM's bandwidth consumption scales with increasing numbers of threads; these increased bandwidth values do not come with substantially increased execution times due to the fact that STREAM is time-limited as opposed to open-ended (see below).

Table 4.4: Average STREAM Bandwidths in MB/s for Local and Remote Execution

Number of Threads	Local Execution	Remote Execution
1	3142.370	2958.292
2	6090.115	5720.102
4	11545.212	10830.317
6	17070.650	15085.381

STREAM was modified in four ways. First, to minimize artificial inflation of the read bandwidth, the number of writes it performs are minimized. All mathematical calculations are performed as normal, but the results are not stored in the destination matrix. Second, as with the matrix multiply benchmark, `mprotect` is used to set the two source matrices as read-only to allow them to be eligible for duplication. Third, STREAM changes which matrix is used as the destination in between sets of operations. To prevent having to contend with changing source matrices, the benchmark was modified so that the two sources are constant throughout the entirety of each STREAM run. Finally, STREAM was given an enforced lower bound on runtime of 100 seconds. Due to the high performance of the machine used to run the experiments, runs would not always reach a consistent steady state in terms of bandwidth. If a STREAM iteration completes before this time boundary is reached, the benchmark starts its computationally-intensive code over. The number of iterations completed and the exact time needed to complete them are stored to allow meaningful statistics about each group of experiments to be calculated.

Finally, as with the matrix multiplication, the data set sizes in STREAM had to be properly configured due to `mprotect` only operating on page-sized sections of memory. Since

the high-performance memory that is being simulated is 1 GB in size (see Section 4.3), the read-only set size in STREAM should be as close to that value as possible without exceeding it. An array size of 48 000 000 elements yields 768 000 000 bytes in the read-only matrices, which is sufficient for experimental purposes.

STREAM performs four calculations[15] multiple times during each iteration, even with the modifications described above. Table 4.5 describes these functions with the following variables: a, b, and c are source matrices. q is a constant scalar. With the modifications made to ensure operability with `mtier`, d is added as the temporary storage variable and the c matrix is unused.

Table 4.5: STREAM Calculations Before and After Modification

Calculation	STREAM Default	Modified STREAM
Copy	$a[i] = b[i]$	$d = a[i]$
Scale	$a[i] = q * b[i]$	$d = q * b[i]$
Sum	$a[i] = b[i] + c[i]$	$d = a[i] + b[i]$
Triad	$a[i] = b[i] + q * c[i]$	$d = a[i] + q * b[i]$

STREAM functions before and after `mtier`-compatible modifications.

4.2.3 STREAM with Validation

Due to the write-minimization described in Section 4.2.2, a STREAM variant was created that performs error checking while the `mtier` module is used. This variant stores an extra copy of each source matrix that is not subject to migration and performs each mathematical operation twice: once using memory that the `mtier` module is managing, and again using the static source matrix copies. If the results of the two calculations do not match, it increments an error counter. The total number of errors counted is reported when the process terminates. Error validation doubles the work performed by the STREAM benchmark; therefore, performance is poor and this variant is used solely for validation and not performance evaluation. With the exception of the extra verification step, this version of STREAM's mathematical behavior is identical to that for the modified STREAM benchmark (see Table 4.5).

The motivation for an error-checking variant of STREAM arose from a serious concern present during the early stages of the work presented in this thesis. Due to the way `mtier`

manages memory, the possibility exists that "stale" memory will be accessed if a page table and translation lookaside buffer become out-of-sync. Identification of errors allowed issues with the module to be corrected during development and provided valuable insight into the overall feasibility of `mtier`'s memory management strategies.

4.2.4 Benchmark Comparison

Tables 4.3 and 4.4 should demonstrate the superiority of STREAM: it is easily able to drive large bandwidth consumption without increasing overall execution times. While the matrix multiplication benchmark is very flexible because it allows for variable set sizes, it is not memory- or bandwidth-intensive enough to be useful when using small matrices and it takes prohibitively long times to run as matrix size (and therefore resident set size and memory bandwidth) increases. While very large set sizes were able to drive larger amounts of bandwidth consumption, it never approached STREAM's maximum and, as shown in Table 4.2, these set sizes would have taken enormous amounts of time to complete. Its poor bandwidth utilization with small set sizes, on the other hand, makes it difficult to draw conclusions about the actual performance of the `mtier` module, as it can be difficult to distinguish low-bandwidth process activity from the slightly-variable background activity that is always present on a memory node. Because of this, while the matrix multiplication benchmark is useful in testing functionality and examining the raw timing effects of migration and duplication, it is not useful for analyzing overall performance effects of the `mtier` framework. Therefore, STREAM is used primarily to analyze the effects of migration and duplication on overall node bandwidth and memory utilization.

4.3 Methodology

For purposes of the experiment, local and non-local NUMA nodes are used. Since traffic on the test machine must cross the QuickPath Interconnect (QPI) to go from one NUMA node to another, a slowdown can in effect be enforced by binding a process' memory to one NUMA

node and its execution to another[3]. Therefore, each process begins with its memory bound to one node and execution bound to another and, when memory is moved to the simulated high-performance memory, the memory is in fact moved to the executing NUMA node. QPI has an effective bandwidth of about 25.6 GB/s[9], so applications that are able to utilize more bandwidth than this should show a greater effect from having memory moved to the executing node.

Table 4.6: Effect of Node Binding on Process Performance

Benchmark	Local	Non-Local	Non-Local / Local
1024 x 1024 Matrix Multiply	10.203 s	10.299 s	1.01
2048 x 2048 Matrix Multiply	257.107 s	553.634 s	215.3
STREAM (One Thread)	7.328 s/iter	7.828 s/iter	1.07
STREAM (Six Threads)	1.323 s/iter	1.502 s/ite	113.5

A number of configurations were used for the `mtier` module and the STREAM benchmark to account for as many variables as possible. The matrix multiplication benchmark has fewer configuration options and therefore is only used with 1024 x 1024 and 2048 x 2048 source matrix sizes. Larger sizes were considered, but exhibited prohibitively long run times and less-than-useful experimental data due to substantially lower bandwidth consumption than the STREAM benchmark).

4.3.1 Baseline Configurations

Two baseline configurations were run for each benchmark. `baseline-local` examines performance when memory allocations and execution are bound to the same NUMA node, while `baseline-remote` examines performance when memory allocations are bound to one NUMA node and execution bound to the other. Baseline data is gathered with the `mtier` module unloaded, although for consistency the processes are executed as tiering-eligible processes. In practice this has no effect on their performance.

[3]The `numactl` command's `membind` and `cpunodebind` flags are used to force memory and execution binding.

4.3.2 `mtier` Configurations

The `mtier` module allows high-performance memory capacity, epoch length, percentage of the high-performance memory to fill each iteration, and frequency of bookkeeping iterations (see Section 3.4.4) to be tuned. To reduce less meaningful variables, only high-performance memory capacity and epoch length are changed during experimental runs. High-performance memory capacities of 12.5%, 25%, 50%, and >100% of the benchmark's resident set size are used (for STREAM, this works out to 98, 196, 392, and 1000 MB, respectively). Epoch lengths of 50, 100, 500, and 1000 milliseconds were used. This yields 16 `mtier` configurations for each benchmark per `mtier` variant (`base`, `heavy`, and `massive`), resulting in 48 total `mtier` configurations.

4.3.3 STREAM Configurations

The number of threads used by STREAM is the only meaningful variable that can be tuned. Because HyperThreading was disabled (see Section 4.1) and because benchmark isolation needed to be isolated to a single socket for results to be meaningful, the maximum number of threads used was six. Single-threaded operation was also examined, as were two- and four-threaded operation. This results in a total of four STREAM configurations to be used for each `mtier` configuration, for a total of 192 `mtier`-managed STREAM configurations and eight baseline configurations. The userspace-only STREAM variant allows variables to be tuned that cause the benchmark to mimic the behavior of the basic STREAM benchmark run with various `mtier` configurations. It was run with the same high-performance memory sizes and thread numbers as the base STREAM benchmark, resulting in an additional 192 experiments.

4.3.4 The Experiment Script

A Python script was written to automate most of the experiments. It handles the loading and unloading of the module in between runs to establish a "clean slate" for the next experiment as well as configuring `mtier`'s parameters. This script also performs analysis of the experiment results when finished, removing possible sources of human error. Since the

simulated fast-tier size is hardcoded, `mtier` must be recompiled every time that paramter changes. `stream_mbind`, must be recompiled between each run due to many of its parameters (epoch lengh and percentage of pages to move in particular) being hard coded.

Chapter 5

Evaluation

5.1 Feasibility

The `mtier_dev_full` (see section 3.8.1) development configuration shows that epoch-based movement of pages in a process' address space is possible without destabilizing the operating system or causing issues with any running processes. The `mtier_dev_partial` (see section 3.8.2) further shows that pages can be targeted for movement (as opposed to migrating whole process address spaces) using slightly-modified default kernel features without causing additional problems. No data was collected from either of these two configurations; their functioning confirms solely the feasibility of continuing to explore this work and they were not fully functional.

5.2 Performance

Once feasibiliy had been established, performance was the next area of exploration. Using the experimental configurations described in Section 3.7, a large number of experiments were run to analyze the usefulness of the memory management scheme employed by the `mtier` module. While some configurations do not result in improvements, many configurations do, and all are better than a naive userspace-only approach.

5.2.1 Frequency of TLB Shootdowns

When starting this work, one of the more pressing questions was the frequency of TLB shootdowns during page migration and the effect these shootdowns had on performance. While the `mtier_base` configuration did not completely elmimate TLB shootdowns, they were reduced by several orders of magnitude. Figure 5.1 shows this reduction over ten STREAM runs (roughly 1000 seconds) using six threads (TLB shootdowns per core were effectively evenly distributed over the six cores).

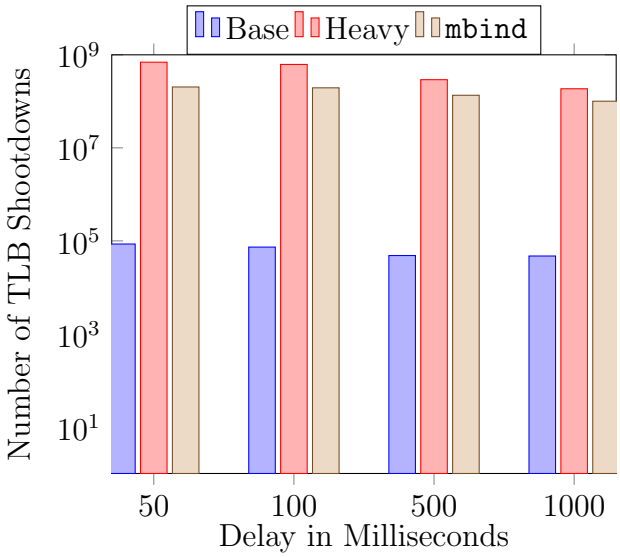

Figure 5.1: Raw number of shootdowns with a 98MB high-performance memory size using varying delays and six threads.

TLB shootdowns in Linux can be configured to flush a single page or all the pages stored in a core's TLB. Different options do exist in the function calls, such as flushing a page range or flushing all the page's of a process' `mm_struct`, but in practice these will fall back to either a number of individual page flushes or a complete core TLB flush due to the capabilities of the underlying hardware. On x86-based architectures, the `INVLPG` instruction is used to invalidate individual TLB entries, while reloading the `cr3` register (even with the same value) is used to invalidate a core's entire TLB.

Most TLB flushes during normal operation consist of context switches, since the `cr3` register contents are changed as a matter of course during switches from one process to another. Explicit TLB flushes, however, consist of two parts. The core that is issuing the

context flush issues an inter-process interrupt (IPI) to all other cores that must flush all or part of the their TLBs. These cores receive this interrupt, suspend any running process, and then perform the requested flush. Since `mtier`'s worker thread is operating on a core independently of the benchmark workloads for the experiments presented in this paper, the IPI sends and serviced interrupts can be tracked easily. Figure 5.2 shows that each core services almost every IPI it receives from the worker thread; in other words, almost every flush of the TLB during page migration, using either default kernel functions or `mtier`, must stop the process on all six execution cores, resulting in a complete halt of any work being performed until the interrupt is finished. Please note that context switches, by definition, also result in the process being temporarily stopped and result in a TLB flush, but are not counted in this figure as they are a matter of routine and do not have any bearing on experimental findings. This figure examines ten six-thread STREAM iterations running with `mtier_base` with a 98MB high-performance memory size and a 50 millisecond delay.

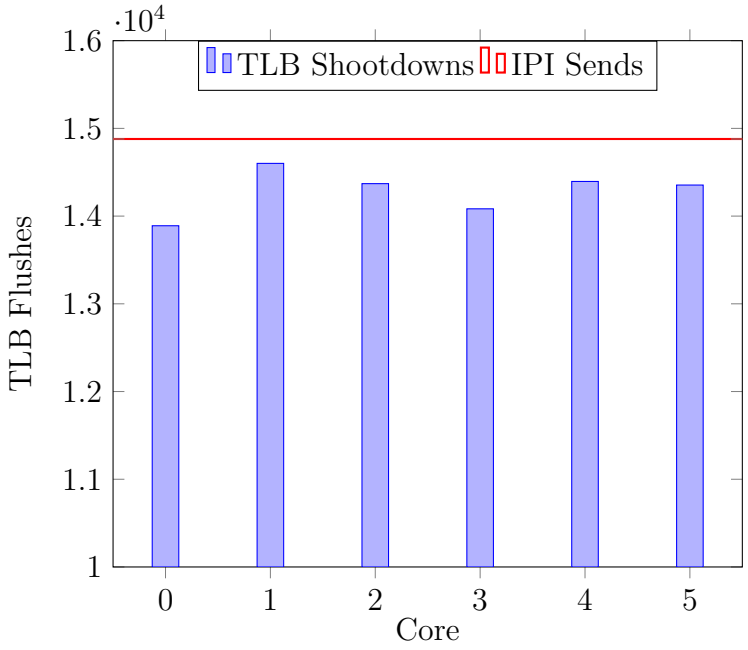

Figure 5.2: TLB Shootdowns per core and total number of IPI sends

For the same workload with varying numbers of threads, Figure 5.3 shows the ratio of total shootdown interrupts serviced to IPIs sent. It demonstrates that almost every IPI results in stoppages on every execution core in use by the workload, which requires a barrier and a complete stoppage of work on all cores until the interrupts have been serviced.

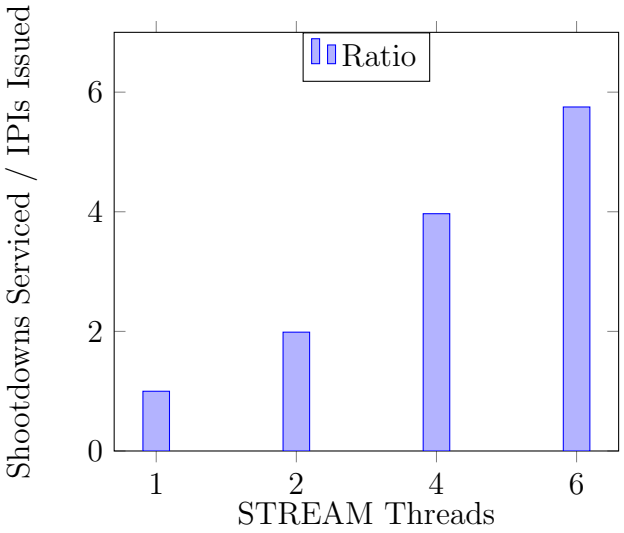

Figure 5.3: TLB Shootdowns per core and total number of IPI sends

The overhead figures for this process range from "several hundreds of [CPU] cycles" for the IPI send alone[4] (about 100 nanoseconds minimum on modern CPUs) to three microseconds[17]. Even using the low estimate, this is non-negligible overhead as the number of TLB flushes increases. The low estimate is an extremely optimistic one since the IPI send is only one part of a multi-step process to flush a TLB entry[17]; however, using the high estimate, overhead quickly becomes crippling. Table 5.1 attempts to quantify the time lost to TLB flushes over ten six-thread STREAM runs (roughly 1000 seconds) using three of the four experimental `mtier` configurations (`mtier_massive` has been omitted because its results are very similar to `mtier_heavy`) with a 98 MB high-performance memory size and 50 millisecond delay.

Table 5.1: TLB Shootdown Time Range

Configuration	Shootdowns	Overhead (100ns)	Overhead (3µs)
Base	85396	.01 s	.25 s
Heavy	694472364	69.43 s	2083.4 s
`mbind`	203591624	20.36 s	678.67 s

The most obvious takeaway from this table is that the three µs overhead figure from 2011 is no longer correct, or is at least not correct with the STREAM workload. The actual overhead must be closer to the 100 ns estimate: to ensure consistecy, the STREAM runs are limited to 100 seconds of clock time, not execution time. This allows measurement of delays

that may be masked if only execution time is measured, but also provides some constraints; for example, the process may not spend more time handling TLB shootdowns than it does actually running. The table also shows that, at a bare minimum, roughly 7% of the process' performance is lost using the default kernel page migration tools, compared to a minimum of a fraction of a percent using `mtier`. While the true overhead is probably greater than this, the service time per shootdown is unchanged for the `base` and `heavy` `mtier` variants; the reduction in overhead comes solely from reducing the total number of shootdowns.

Figure 5.4 shows just how large this reduction is when using `mtier` with a 98MB high-performance memory size and 50 millisecond epoch to manage memory for ten six-thread STREAM runs. The number of shootdowns is the core average and not the grand total. `mtier_heavy`, which uses built-in kernel migration tools, has almost one shootdown per page that is managed. `mtier_base`, however, manages tens of thousands of pages per shootdown. Interestingly, each individual shootdown in both variants is an `mm_struct` shootdown (in other words, the entire process address space is flushed from the CPU; whether this happens with a number of `INVLPG` calls or a `cr3` register flush and reload is up to the operating system and not relevant). For the `mtier_base` variant this makes sense since large numbers of page mappings are changed prior to each flush. However, for the `mtier_heavy` variant, the almost-1:1 ratio of pages managed to flushes performed suggests that there is no coalescing of TLB flushes and that large performance penalties are incurred performing excessive core-wide TLB flushes.

5.2.2 Migration and Movement Performance

The performance impact of default kernel migration faculties has been described as insignificant[7]; however, frequent and targeted migration of pages has not been well-explored. The exact impact of page migration was a major question in this work, as was the possibility of reducting the time needed to perform certain types of memory management. Types of memory movements have been categorized as "heavy copies," which are `mtier`-specific copies that involve movement of both a page's contents and its mapping, "swaps," which are `mtier`-specific changes in page mappings from a page in high-performance memory to an existing copy of the same page in low-performance memory, "kernel-style migrations,"

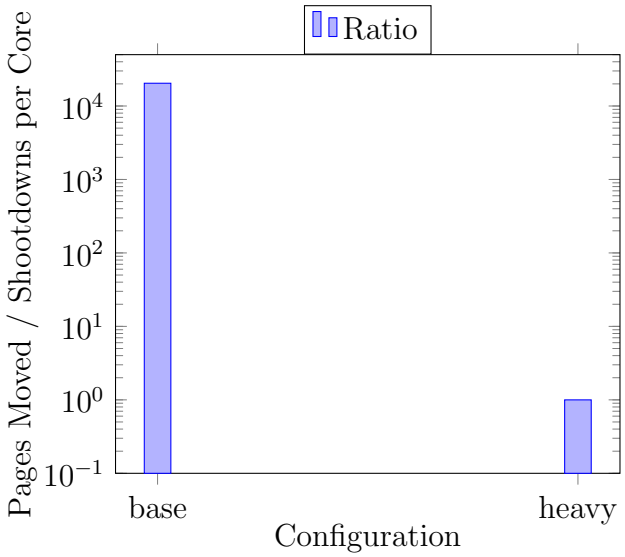

Figure 5.4: Ratio of pages moved to shootdowns per core by `mtier` variant

used by `mtier_heavy` in which a page is moved to a pre-allocated page on a differeny memory tier, and "kernel-style alloc migrations," used by `mtier_massive` in which a page is moved to a different tier of memory, but its destination page is not pre-allocated. Table 5.2 shows exactly what is required for each type of migration.

Table 5.2: Comparison of Migration Types

Type	Contents Copied	TLB Flush	Allocation Required
Heavy Copy	Yes	No	No
Swap	No	Yes	No
Kernel	Yes	Yes	No
Kernel w/ Alloc	Yes	Yes	Yes

Taken at face value, heavy copies and swaps should have the lowest performance requirement. Heavy copies do not require a TLB flush; instead, because the page is left intact on the low-performance tier, the mapping can be changed and the process can continue accessing the old mapping until a TLB flush happens organically (e.g. from a context switch). Swaps, on the other hand, do not require page contents to be copied but do require an explicit TLB flush because the high-performance mapping could be immediately invalidated. However, because `mtier`'s behavior is predictable (i.e. swaps happen in bunches at the same time prior to additional heavy copies that may replace pages on the fast tier), a single TLB flush will suffice for potentially a large number of swaps. Table 5.5 shows

this to be the case. `mtier`'s fast PTE swaps (A) are over 22 times faster than the kernel's built-in migration functions (C), while `mtier`'s heavier copy with PTE swap (B) are 2.6 times faster. The kernel migration functions that employ page allocation and freeing (D) are almost equivalent to the kernel's migration functions using pre-allocated memory. This explains why `mtier_heavy` and `mtier_massive` have virtually identical performance values, which is why `mtier_massive` is not included in any other figures in this thesis.

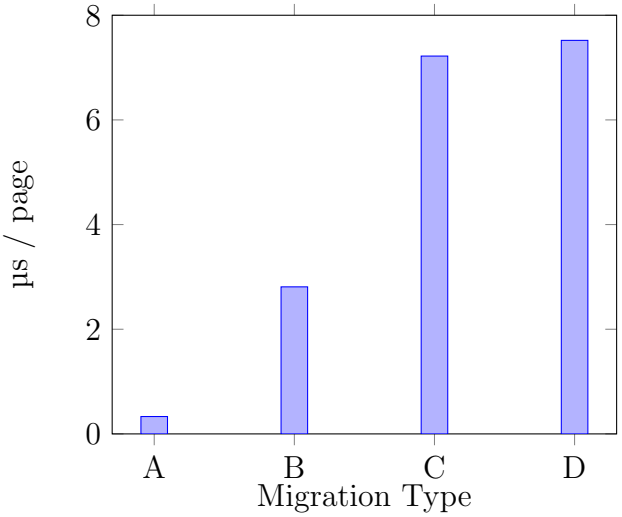

Figure 5.5: Completion times for different types of page movements

These two reductions in migration time are substantial. However, this is not the whole story. Some parts of the migration process for kernel-style migrations are able to occur in the background without suspending the process for which the migration is being performed. Therefore, this time is not indicative of the time the process is completely blocked during migration, but due to the way the kernel[16] and the architecture[11] are designed, blocking does occur at least during the TLB shootdown and data copy stages.

5.3 Effect of Epoch Time on Performance

Intuition would suggest that longer epochs would result in better performance, and this is true for `mtier`, although the difference is marginal at best. With all epoch lengths, `mtier` exceeded the performance of the remote baseline, while `mtier_heavy` only approached this performance for the longest epochs and `stream_mbind` was consistently much worse and

never reached the remote-only baseline performance. Figure 5.6 quantifies this performance with ten six-thread STREAM runs (roughly 1000 seconds of runtime) running on `mtier`, `mtier_heavy`, and `stream_mbind` with a 25% STREAM RSS simulated fast tier size and varying epochs. All values are normalized to a percentage of the local execution baseline. The dashed line represents the remote execution baseline normalized to a percentage of local execution. Interestingly, considering that `mtier`'s dataset is never fully on the fast tier, its execution time suggests that almost any overhead resulting from migration operations is removed, although the improved performance as epoch length increases shows that not all of it can be completely eradicated.

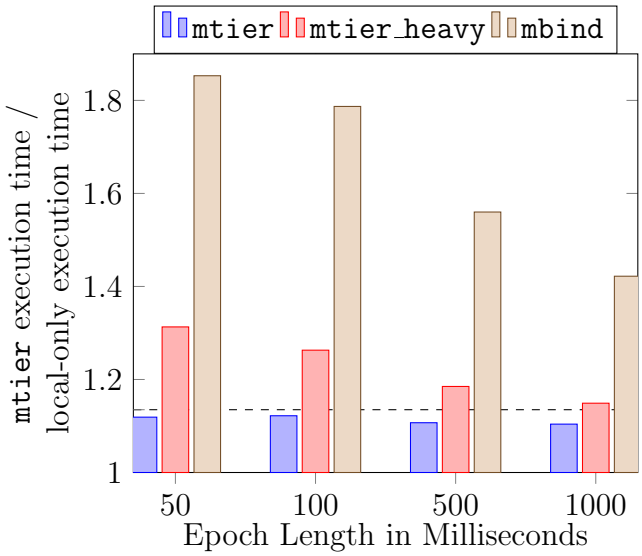

Figure 5.6: Effect of Epoch Length on Performance. Lower is better.

One side effect of short epochs noticed during this batch of experiments is that `mtier_heavy` is unable to complete as many total iterations as `mtier` due to the increased migration operation times for the default kernel functions quantified in Figure 5.5. This means that, in addition to being able to exceed `mtier_heavy`'s performance, `mtier` is able to migrate more pages in the same amount of time, which helps put the reduction of TLB shootdowns outlined in Figure 5.1 into better perspective. Figure 5.7 shows the ratio of successfully completed epochs in `mtier_heavy` to those in `mtier`.

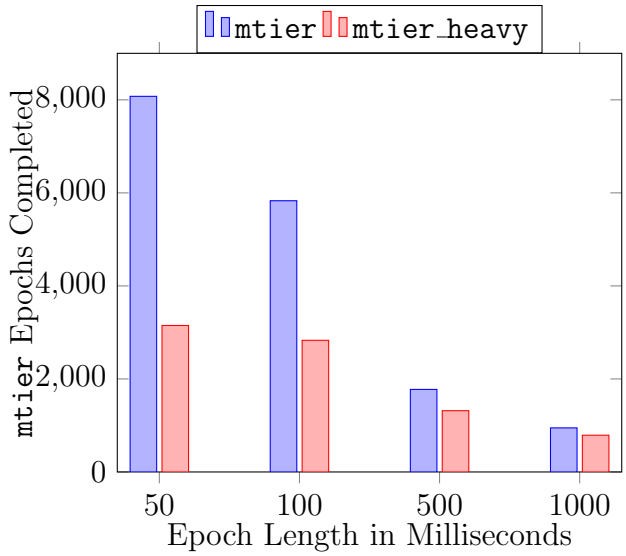

Figure 5.7: Effect of migration function completion times on number of epochs completed in 1000 seconds

5.4 Effect of Simulated Fast Tier Size on Performance

The size of the high-performance memory relative to the benchmark's RSS had remarkably little effect with `mtier` and a larger effect with `mtier_heavy`. `mbind` results are still outstanding. This can be explained by the overall reduction of overhead in `mtier` and the longer times `mtier_heavy` spends blocking the executing process. As with the epoch time benchmark, `mtier` consistently exceeds the remote-only execution time and in some cases behaves as if there is almost no visible overhead, while `mtier_heavy` is consistently worse and `stream_mbind` is substantially worse. Unlike the previous experiment, `mtier_heavy`'s performance improves dramatically as the size of the simulated fast tier increases and never meets the remote execution baseline performance. As with Figure 5.6, all values are normalized to a percentage of local execution time and the dashed line represents the remote execution baseline as a percentage of the local execution baseline. Shown is a ten-run, six-thread STREAM experiment with `mtier` configured to have a 50 millisecond epoch and varying high-performance memory sizes.

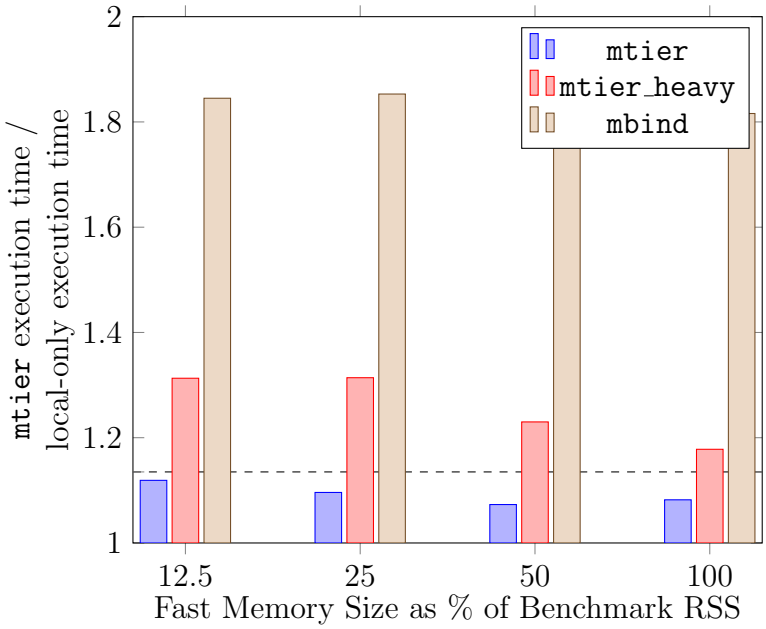

Figure 5.8: Effect of high-performance memory size on performance. Lower is better.

5.5 Effect of Number of Threads on Performance

Applications managed by `mtier`, as would seem logical, benefit, sometimes greatly, from increased numbers of threads, to a point. As shown in Figure 5.9, in all cases, `mtier's` performance is better than that of `mtier_heavy`, although there seems to be a decrease in relative performance with six threads. This could be a fluke, or it could be a trend that would continue to be detrimental to performance as the number of threads or execution cores increases; access to a machine with more physical execution cores would allow this to be confirmed. As shown in Figure 5.10, `mtier`'s absolute performance continues to improve as the number of threads is increased, Figure 5.9 just shows that its relative benefit when compared to all-local execution decreases somewhat.

The userspace solution, `stream_mbind` behaved unpredictably as the number of threads increased and is therefore not shown in either of these figures. Its data suggests that it actually performs better with smaller numbers of threads, but the results were so varied between runs that no conclusion is being drawn from the data that is available.

Figure 5.10 shows raw execution times for STREAM iterations with varying numbers of threads. Each benchmark or baseline shows an absolute improvement in performance as the

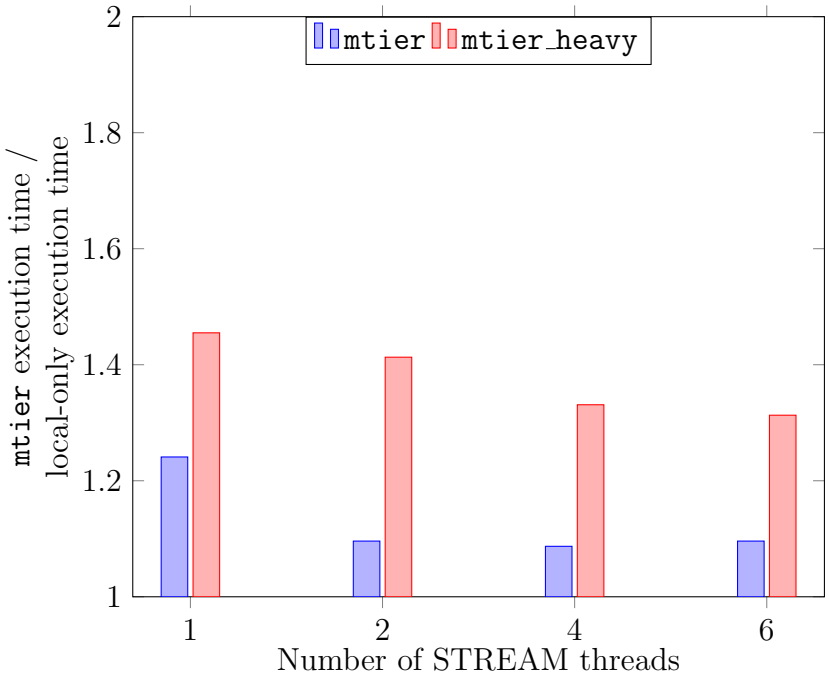

Figure 5.9: Effect of number of threads on performance for an `mtier` configuration with a 50 ms epoch as 12.5% benchmark RSS fast memory size. Values normalized to percentage of local execution baseline for respective number of threads.

number of threads increases. `mtier`'s performance is worse than even the remote baseline for all but the greatest number of threads. `mtier_heavy` performs worse than either baseline for all possible numbers of threads. Clearly `mtier` is not a good solution for single-threaded applications; however, more physical threads will be needed in the future to determine the long-term behavior of the trends visible in these two figures.

5.6 Analysis of `mtier`

`mtier` performs admirably in all measured use cases. It is capable of exceeding the performance of the remote baseline in all measured configurations and in some cases approaches its ideal (zero overhead) performance for multi-threaded applications. For single-threaded applications it fares worse; this suggests that the overhead that remains can be partially overcome by high levels of parallelism and that `mtier` would be more useful in in the types of high-performance applications hybrid architecture machines would be expected to run than in applications in which parallelism is poor or nonexistent. Since page shadowing

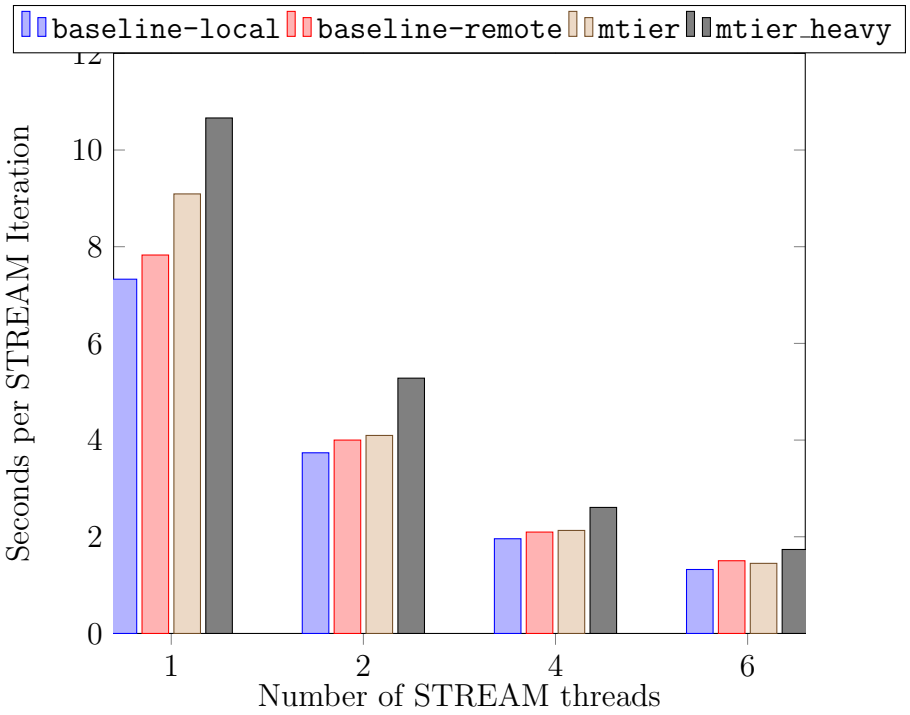

Figure 5.10: Effect of number of threads on performance for an `mtier` configuration with a 50 ms epoch as 12.5% benchmark RSS fast memory size. Values normalized to percentage of local execution baseline for respective number of threads. This graph shows raw performance values.

and TLB shootdown batching are the only two performance improvements present in `mtier` and not present in `mtier_heavy`, the only logical conclusion is that these two features are responsible for the large performance difference between the two benchmarks.

5.7 Analysis of `mtier_heavy`

`mtier_heavy` fares worse than `mtier`. In some cases it approaches the remote execution baseline performance values for a given configuration, but it never exceeds it. While it is still better with larger numbers of threads, its overall poor performance compared to the alternative implies that it would not be a viable method of managing memory in a hybrid architecture for any application; the user would be better off just binding all their allocations to the low-performance memory. The differences in performance between `mtier` and `mtier_heavy` provide valuable insight into the performance overhead incurred when frequent TLB shootdowns and data copying are present. Previous tools have been able to

overcome this by not allowing policy or memory binding to be readily changed when an application is running[6, 7, 12], but this comes at a cost of inflexibility with the memory manager itself.

`mtier_massive`, which is not represented in any previous figures due to the similarity of its results and the results of `mtier_heavy`, shows that page allocation overhead is fairly insignificant when compared to the overhead of TLB shootdowns and data copies; therefore, `mtier`'s pre-allocation of pages does not provide a substantial performance improvement and could probably be safely removed from the module (resulting in a page allocation when data is promoted to high-performance memory and a page free when a page is demoted to low-performance memory), which could provide some more flexibility with the module and better coexistence of the module and operating system.

5.8 Analysis of Userspace Solution

The userspace solution evaluated in `stream_mbind` fares the worst of all three evaluated benchmarks. Its performance is substantially worse than either `mtier` variant for all measured configurations, although its performance improvement as epoch length and fast memory size increase are much more dramatic. At best, its execution times are 140% of the local execution baseline and are always dramatically worse than the remote execution baseline.

This can be explained by a phenomenon that occurs in userspace that would not affect either module, both of which run in kernel mode. The use of system calls, therefore, introduces two extra sources of overhead. The first is the system call itself which involves looking up a function in a jump table and calling it[16], which is never as efficient as calling a function directly. The second comes from the initial privilege mode switch when the CPU is switched from nonprivileged to privileged execution mode[11].

This poor performance coupled with the wide-ranging changes to the process itself required to implement a userspace solution shows that it is effectively useless to perform this type of management without, at the very least, a kernel module; the returns will never outweigh the cost of the modifications themselves and the new overhead.

Chapter 6

Future Work

While the version of `mtier` presented in this thesis is sufficient for experimental work, additional work is required for it to become a production-ready tool.

6.1 Module-Only Framework

By moving all `mtier` functionality into the kernel module, it can be made a kernel-agnostic framework. Users would have to provide a COMPATIBLE kernel, but they would be able to download a default mainline distribution as opposed to a customized kernel. This would allow for more widespride availability of the framework as well as increased portability. This is arguably the most trivial area of future work required and also the most important if `mtier` is to be distributed as a self-contained kernel module.

6.2 Improved Page Eligibility Determination

The experimental version of `mtier` determines page eligibility for migration by looking for pages that have been explicitly marked read-only. A runtime profiling technique like that in `Carrefour`[6] would be able to effectively determine which pages are *most likely* read-only and therefore eligible for duplication. Due to the ever-present risk of writable pages being selected using this profiling technique, write protection (see section 6.3 would be needed to make this work most effectively.

From an implementation standpoint, `mtier` only needs a linked list of `struct page` pointers linked on the `LRU` member, so in theory it could even be designed to allow users to supply their own eligibility-determination functions. However, this would be a fairly advanced feature and should be included *in addition to* a generic "default" function, not *instead of* it.

Ideally, `mtier` will never require an expensive pre-run profiling step like X-Mem[7]. Static profiling removes both flexibility and user-friendliness.

6.3 Write Protection

In its current form, `mtier` suffers from synchronization issues if writable pages are somehow included in its eligible page list. When a page is promoted to high-performance memory, its slow copy is still accessed until an organic context switch occurs. Conversely, when a page in fast memory is demoted back to slow memory, its fast copy is still accessed until the TLB shootdown for the current batch of evictions is processed (see chapter 3 for a more detailed explanation of this behavior). In either case, when the TLB shootdown DOES occur, the fast and slow pages could be out-of-sync with each other, resulting in issues with the running process.

Taking advantage of certain hardware features, particularly the dirty bit in the page table entry[11], could alleviate this issue somewhat. By clearing the dirty bit before performing migration operations and checking it after the operations are performed, writes can be detected and the migration operation can be invalidated before it is finalized. This would induce a penalty to `mtier`'s performance but it should not be substantial as long as only a small minority of of migration operations ever have to be invalidated.

6.4 Managed Language Support

`mtier`'s support for languages like Java and Python are virtually nonexistent. `mtier` is designed from the get-go to support native executables with clearly-marked heaps and virtual memory areas in the style expected by the operating system[16]. While modifications to the

languages' virtual machines would most likely be needed to make this work, the general techniques used by `mtier` could most likely be adapted with some effort, and doing so would increase the availability and overall usefulness of the module substantially. Due to the fact that it is open source, the Hotspot Virtual Machine for Java[1] is an obvious first candidate for managed language support.

6.5 `MulPTE`

`MulPTE` (pronounced "multi") is a radical departure from `mtier` and would stand by itself as opposed to being a modification or addition to the `mtier` framework. It is a proposed modification that would require hardware as well as software support. By allowing CPUs and operating systems to recognize and manipulate "wider" page table entries, it may be possible to have much more reliable and performant backing stores within byte-addressible memory. From reading the *Intel Software Developers' Manual*[11], I have concluded that there are free bits within the page table entry that could be leveraged for a task like this, but that would also be completely backwards-compatible with older hardware and software.

One possibility would be to have `MulPTE` support single-, double-, and quadruple-width page table entries, therefore allowing it to map a hybrid memory system of up to four tiers. Existing dirty bits could be used to identify tiers that have been written to and therefore require synchronization, while spare bits could be used to identify each tier's memory type (e.g. "DRAM," "NVM," and "High-Performance." Finally, additional free bits could be used to identify the tier currently in use. This would monopolize most of the free bits in a page table entry, but could theoretically be much faster than software-managed hybrid memory, while retaining the same level of flexibility (depending on the operating system).

One disadvantage of `MulPTE` is that it would result in increased page table sizes for allocations that are eligible to use hybrid memory. Using the proposed quadruple-width page table entries described above, page table density would be reduced by a factor of four. By tailoring which allocations are eligible to use hybrid memory, either through special allocation functions available to developers, a profiling run, or some sort of runtime analysis, the actual increase in page table size would have an upper limit of 400% but would most likely be less.

While testing would be required to confirm the effectiveness of this approach, it is entirely possible that, for high-performance applications, this increase in memory consumption would be a small price to pay for the potential performance benefits.

6.6 More Exotic Solutions - TLB Modifications

The design of the translation lookaside buffer is not conducive to hybrid memory management. TLB flushes are expensive in terms of execution time. If a page mapping changes, there is no option to update any TLBs[11]: they must be flushed either one-page-at-a-time in a loop, or in their entireties. TLBs could be modified to be directly compatible with something like MulPTE (see Section 6.5) in the form of an extra-wide TLB line with supporting instructions, or an instruction could be added to the architecture to allow the updating of an existing TLB line. Either way, the current process of erasing the TLB line and waiting for it to repopulate through some sort of access would be circumvented, and the TLB could be updated directly via a CPU instruction. This could reduce the migration overhead from microseconds to handfuls of CPU cycles (depending on the actual implementation), and would even benefit existing NUMA systems in which the migration process requires similar manipulation of the TLB. This would be a fairly involved architectural undertaking and its actual implementation is beyond the scope of this thesis.

Clearly, there are many improvements that can be made to the `mtier` module as it is presented in this thesis, operating system memory managers, and even hardware to improve management of emerging hybrid or even existing non-uniform memory architectures. While it shows potential as a capable and effective system for managing hybrid memory systems, it has not reached a stage of development that would allow it to be used with production systems. The best order for implementing these changes is presented above, with `MulPTE` coming last, if at all, or being handled by a more hardware-centric research group.

Chapter 7

Conclusion

Memory management in hybrid memory systems is an under-studied field. While it is not entirely dissimilar to memory management in NUMA systems and tools like `Carrefour`[6] and `mcolor`[12] can be used as starting points and inspiration, even dedicated tools like `X-Mem`[7] do not address any reduction in overhead of migration operations or equitable distribution of higher-performance memory in hybrid systems.

This thesis has presented the `mtier` framework and a proposed method of runtime memory management in hybrid memory systems. By operating on read only memory (with the assumption that, in production systems, that most memory objects are effectively read-only following initialization[3]), `mtier` is able to use page shadowing and TLB shootdown batching to significantly reduce – in some cases, almost eliminate – overhead from page migration operations between DRAM and a simulated high-performance memory tier. By doing so, `mtier` has demonstrated that these are effective memory management techniques and lays the groundwork for future research into the management of memory in emerging hybrid architectures.

Bibliography

[1] (2018). Hotspot group. http://openjdk.java.net/groups/hotspot/. 52

[2] (2018). Intel optane technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html. 4

[3] Akram, S., Sartor, J. B., McKinley, K. S., and Eeckhout, L. (2018). Write-rationing garbage collection for hybrid memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 62–77. 2, 5, 10, 54

[4] Amit, N. (2017). Optimizing the tlb shootdown algorithm with page access tracking. In *2017 USENIX Annual Technical Conference*, USENIX ATC '17, pages 27–39, Santa Clara, CA. USENIX Association. 8, 40

[5] Atwood, J. (2007). The danger of naivete. Online at https://blog.codinghorror.com/the-danger-of-naivete/, last accessed 30 May 2018. xii, 17, 61

[6] Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., and Roth, M. (2013). Traffic management: A holistic approach to memory placement on numa systems. *SIGARCH Comput. Archit. News*, 41(1):381–394. 7, 49, 50, 54

[7] Dulloor, S. R., Roy, A., Zhao, Z., Sundaram, N., Satish, N., Sankaran, R., Jackson, J., and Schwan, K. (2016). Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 15:1–15:16, New York, NY, USA. ACM. 6, 41, 49, 51, 54

[8] Foundation, F. S. (2008). *mbind Linux User's Manual*. Linux. 6

[9] Intel Corporation QPI Introduction (2009). *An Introduction to the Intel QuickPath Interconnect*. Intel Corporation. 28, 34

[10] Intel MCDRAM Overview (2016). *An Introduction to MCDRAM (High Bandwidth Memory) on Knights Landing*. Intel Corporation. 4, 6

[11] Intel Software Developer's Manual vol. 3 (2018). *Intel 64 and IA-32 Software Developer's Manual, Volume 3 (System Programming Guide)*. Intel Corporation. 7, 43, 49, 51, 52, 53

[12] Jantz, M. R., Strickland, C., Kumar, K., Dimitrov, M., and Doshi, K. A. (2013). A framework for application guidance in virtual memory systems. In *ACM SIGPLAN Conference on Virtual Execution Environments*, VEE '13, pages 155–165. 7, 49, 54

[13] Kleen, A. (2004). *numactl(8) Linux User's Manual*. SuSE Labs. 5

[14] Labs, S. (2007). *numa Linux User's Manual*. SuSE Labs. ix, 6, 59

[15] Raman, K. (2013). *Optimizing Memory Bandwidth with Stream Triad*. Intel Corporation. 30, 32

[16] Torvalds, L. (2018). Linux (4.4.17)[operating system]. http://www.kernel.org/pub. ix, 43, 49, 51, 59

[17] Villavieja, C., Karakostas, V., Vilanova, L., Etsion, Y., Ramirez, A., Mendelson, A., Navarro, N., Cristal, A., and Unsal, O. S. (2011). Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 340–349. 8, 10, 40

Appendices

A Kernel Functions

The `mtier` framework is dependent on (or was during development) a number of kernel functons. The more complex or obscure ones are described here.

A.1 `migrate_pages`[14]

`migrate_pages` is a function available in stock Linux kernels that allows the movement of all eligible memory in a process' address space to be moved from one NUMA node to another. It handles elgibility-checking, allocation of new pages, freeing of old pages, data and flags copying, and page table modification. It also has a corresponding system call to allow a process to request its own address space be migrated.

Unmodified, `migrate_pages` does not allow any sort of targeted migration (short of selecting where the new memory is located). Process stack, heap, and any eligible memory of other types is moved. Partial migrations are only performed if some pages are not eligible for migration or if the destination NUMA node does not have enough free space to fulfill the request.

A.2 `unmap_and_move`[16]

The `unmap_and_move` function is available in stock Linux kernels. It takes a source page, an allocation function, and two mode flags and will handle the allocation of a new page, copying of the source page flags and data, and page table modification needed to move one page to a different location.

This function is called during the `migrate_pages` function and handles a substantial amount of the actual migration work.

A.3 `mtier_unmap_and_move`

`mtier_unmap_and_move` is the `mtier`-specific version of `unmap_and_move`[16] used for `mtier_heavy` and `mtier_massive`. It handles the same general tasks as the default version of the function, but does not allocate a new page for the migration's destination. Instead, it accepts as

an argument a pointer to a page that has already been allocated. `mtier_massive`, which does not use pre-allocated memory, must therefore handle the allocation explicitly prior to calling this function. This differes from `mtier` and `mtier_heavy`, which both allocate all their simulated fast memory when they are first loaded instead of when the memory is needed.

B Algorithms

B.1 Fisher-Yates Algorithm

The Fisher-Yates shuffle algorithm is employed in `mtier` to quickly "shuffle" the pagelist during execution of standard iterations. In its simplest form, the algorithm is:

```
for(i = n - 1; i > 0; i--) {
    int x = rand[0, i + 1]
    swap(array[i], array[n]);
}
```

Figure 1: Simple pseudocode for the Fisher-Yates shuffle algorithm [5]

Since the pagelist is a linked list, the algorithm in figure 1 had to be modified. The form employed in `mtier` is:

```
int i = 0;
for_each_entry(pagelist) {
    i += 1;
}
arr = allocate(i * sizeof(struct page *)) bytes;
for(j = i - 1; j > 0; j--) {
    int x = rand[0, j + 1]
    arr[j] = pagelist[x];
}
```

Figure 2: Modified version of Fisher-Yates used in `mtier`.

The code in figure 2 is pseudocode, but it should give a good idea of how the shuffling is handled. The pagelist itself is not shuffled, but the pointers to it are, and the shuffled array of pointers can then be iterated over to manipulate randomly-selected pages.

Vita

Joseph Teague was born in Chattanooga, TN, as the only child of Victoria Medaglia and Thomas Teague. After growing up in Lewiston, Maine and Natick, Massachusettes, he returned to East Tennessee to attend Oak Ridge High School. After graduating, he took some time off from academics and worked for the United States Department of Energy in Oak Ridge. At the end of 2012 he received an Associate's Degree from Roane State Community College and accepted a job as a mobile application developer for Science Applications International Corporation. In 2016, Joseph obtained a Bachelor's Degree in Computer Science from the University of Tennessee, Knoxville and, at the encouragement of Dr. Michael Jantz, pursued graduate school following a six month internship with Intel in Chandler, AZ in the fall of 2016. Joseph was awarded a research fellowship prior to starting grad school and has worked as a Graduate Research Assistant and, as needed, a Graduate Teaching Assistant. He received his Master's Degree in Computer Science from the University of Tennessee under the tutelage of Dr. Michael Jantz at the end of summer, 2018 and is continuing to study for his Ph.D. at the same institution under Dr. Michela Taufer.