MODERNIZING OPERATING SYSTEM KERNEL MEMORY MANAGEMENT FOR 1ST-PARTY DATACENTER WORKLOADS

by

Mark Mansi

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2023

Date of final oral examination: April 5, 2023

The dissertation is approved by the following members of the Final Oral Committee:
Michael M. Swift, Professor, Computer Sciences
Andrea C. Arpaci-Dusseau, Professor, Computer Sciences
Mikko Lipasti, Professor, Electrical and Computer Engineering
Jignesh M. Patel, Professor, Computer Sciences
Shivaram Venkataraman, Associate Professor, Computer Sciences

To my parents; they sacrificed much so I would have opportunity.

ACKNOWLEDGMENTS

There are always plenty of rivals to our work. ... The second enemy is frustration—the feeling that we shall not have time to finish. If I say to you that no one has time to finish, that the longest human life leaves a man, in any branch of learning, a beginner, I shall seem to you to be saying something quite academic and theoretical. You would be surprised if you knew how soon one begins to feel the shortness of the tether, of how many things, even in middle life, we have to say "No time for that," "Too late now," and "Not for me." But Nature herself forbids you to share that experience. A more Christian attitude, which can be attained at any age, is that of leaving futurity in God's hands. We may as well, for God will certainly retain it whether we leave it to Him or not. Never, in peace or war, commit your virtue or your happiness to the future. Happy work is best done by the man who takes his long-term plans somewhat lightly and works from moment to moment "as to the Lord." It is only our daily bread that we are encouraged to ask for. The present is the only time in which any duty can be done or any grace received.

— C.S. Lewis, *The Weight of Glory*, 1941

I have often wondered if people actually read Acknowledgements sections. I'm glad that you, dear reader, have chosen to do so. Below is a long, long (yet incomplete) list of people who made this dissertation possible.

As a Christian, I believe that ultimately, all good things come from God – whether that is life itself or the skills to do the work found herein or the long list of people below. First and foremost, my thanks go to him!

Among mortals, the person most directly responsible for my success is my advisor, Mike Swift. When I was searching for grad schools and potential advisors, Mike was recommended to me as an excellent researcher by multiple people. I did not know at the time that he would also turn out to be very patient, generous, supportive, and kind. It's one thing to be an excellent researcher; it's another entirely to be an excellent person and a role model. Mike, thank you for being my advisor for the past several years, with all that entails!

Still, despite a quite enviable position, this journey has been... difficult. I've probably wanted to quit at least once per month (and often much more than that) for the past 6 years. Imposter syndrome, burnout, the stress of constantly failing/broken experiments, boredom, the frustration of not making progress, etc. have all hit me hard (more than once). Through these times, the listening ears, gentle encouragements, and prayers of my parents and sister, Ingolf and Erika Wallow, Josh Camacho, Mark Barnard, Brian and Simone McLoughlin, Jianting He, and Lloyd Biddle have been absolute lifesavers. I think that they might have quite literally saved my sanity. Thank you all so much for the love you've shown me in many ways.

I owe my parents still more thanks for the personal sacrifices they made to get me here. My parents are the hardest working people I've ever known, and so much of that hard work has been directed at making opportunities for my sister and I. They immigrated to the US from a second-world country with two young kids and not that much money, but now both of those kids have gone to good colleges and grad schools without debt. My parents epitomize the American Dream, but they never worked for their own comfort. Mom and Dad, I am so priveleged to have you as my parents.

I've received significant and formative advice from many sources. My sincere thanks go to Mark Hill, whose golden advice on finding a research topic and organizing a presentation have enriched my work. He has also given significant and insightful feedback on my work from early exploration to paper drafts. Thank you also, Swapnil Haria, Ram Alagappan, Aishwarya Ganesan, Mike Marty, Yuvraj Patel, and Lena Olson for hours and hours of discussion, advice, feedback, and insights on a truly staggering array of topics, both

technical and career-related. My thanks also go to my defense committee, Andrea Arpaci-Dusseau, Mikko Lipasti, Jignesh Patel, and Shivaram Venkataraman, who gave insightful criticism that shaped my research. Numerous anonymous paper reviewers from several conferences also gave insightful and formative feedback; though frustrating at times, constructive criticism is the path to growth.

Among all the things I expected to find in grad school, I was not expecting to meet such a wealth of wonderful people – people who were not only good computer scientists, but also friends. I've enjoyed discussions, feedback, projects, ice cream, and snowball fights with many colleagues. Thank you, Bijan Tabatabai, for being an excellent project partner and office mate; working with you has been a great pleasure! Thank you also, Yanfang Le and Anjali, for being great office mates and working late into the night with me. Thank you to my peers in the Multifacet and SCAIL groups, Swapnil, Anjali, Sujay Yadalam, Deepak Sirone, Bijan, Hayden Coffey, Ashwin Poduval, Yusen Liu, Pratyush Mahapatra, Chloe Alverti, and Varun Ravipati for many, many discussions and (last-minute) paper reviews. Thank you Anthony Rebello, Kostas Kanellis, Jing Liu, Archie Abhashkumar, Arjun Singhvi, Vinay Banakar, and Danish Shaikh for many hours of fascinating (sometimes random) discussions, distractions, reading, and meals. Thank you Akhil Guliani, Varun Naik, Karan Bavishi, Hasnain Ali Pirzada, Suhas Pai, and Preyas Shah; our reading groups, project groups, and celebratory meals were a consistent source of both knowledge and laughter for me – until you all graduated (I've missed y'all a lot!). Thank you to David Merrell, Varun Chandrasekaran, and Aman Singh Saini for co-leading SACM with me for many years; I grew and learned so much from that experience and am grateful for our friendship. Thank you to Abhanshu Gupta and Steve Yang for being great TA office mates, being some of the first friends I made in Madison, and teaching me a bit about your cultures. As long as this paragraph is, I'm absolutely certain I've forgotten a few people (if that's you, please accept my humble apologies). Thank you all from the bottom of my heart! I truly hope that our paths may cross again.

Finally, along the way to grad school, I've been blessed with many excellent teachers and professors – more than I could possibly name, but a few stick out. Thank you Betty Wheeler for investing time into a certain lagging (and cheating) 3rd-grader and starting me on the path of academic excellence and integrity. Thank you Calvin Lin for putting up with my (in retrospect) rather bad undergraduate research and encouraging me to go to grad school. Thank you Ahmed Gheith for inspiring a formational love of operating systems in me and getting me two internships; in my 22-year studenting career with numerous excellent teachers, your classes stick out as my favorites and some of the most impactful. In the course of your diligent work, you all have shaped my life forever.

CONTENTS

Contents iv

List of Tables vi

List of Figures vii

Abstract viii

1 Introduction 1
- *1.1 Contributions* 2
- *1.2 Organization of this Dissertation* 4

2 Background 5
- *2.1 Virtual Memory* 5
- *2.2 What does Memory Management do?* 6
- *2.3 Challenges in Modern Memory Management* 9
- *2.4 Target Setting and Scope* 13

3 0sim: Preparing System Software for a World with Terabyte-scale Memories 15
- *3.1 The Capacity Scaling Problem* 17
- *3.2 Design of 0sim* 18
- *3.3 Implementation* 21
- *3.4 Simulator Validation* 25
- *3.5 Data-obliviousness and Speed* 29
- *3.6 Case Studies* 31
- *3.7 Future Work* 36
- *3.8 Conclusion* 36

4 CBMM: Financial Advice for Kernel Memory Managers 38
- *4.1 Evaluating Current Behavior* 40
- *4.2 Cost-Benefit Memory Management* 43
- *4.3 Implementation* 49
- *4.4 Evaluation* 53
- *4.5 Future Work* 62
- *4.6 Conclusion* 63

5 Characterizing Physical Memory Fragmentation 64
- *5.1 Study: Linux Allocation Sizes* 64
- *5.2 Study: Real World Fragmentation* 65
- *5.3 Future Work* 75
- *5.4 Conclusion* 76

6 Experiences and Takeaways 77

7 Conclusions and Future Work 81

Bibliography 84

LIST OF TABLES

2.1 Different MM policies and their goals and pathologies 12

3.1 Specifications of test platforms for 0sim. 25
3.2 Observed aggregate compressibility ratio for various workloads 29
3.3 Time spent and pages scanned and reclaimed with different reclamation policies and modes. . 34
3.4 Number of chunks and time spent initializing and freeing memory during boot with ktask . . . 34

4.1 Description of workloads for CBMM evaluation 53

5.1 Contiguous memory usage of notable processes in various workloads 65
5.2 Description and count of instrumented nodes in fragmentation study 68

LIST OF FIGURES

3.1 Design of 0sim (left) and Simulation State (right). 19
3.2 Example of proposed DTS mechanism in 0sim 23
3.3 Accuracy validation experiments of 0sim on different host machines for single-core workloads . 27
3.4 Accuracy validation experiments of 0sim on different host machines for multi-core workloads . 28
3.5 Simulation duration and slowdown compared to native execution 30
3.6 Amount of memory in each kernel page allocator free list in mix workload 32
3.7 Impact of continuous compaction on latency of memcached requests 35

4.1 Runtime and usermode cycles spent in page walks for different parts of application address spaces 41
4.2 CDF of Linux soft page fault latency by type of page fault 42
4.3 CBMM model inputs and outputs. 45
4.4 CBMM huge page cost-benefit model. 50
4.5 CBMM async prezeroing cost-benefit model 52
4.6 CBMM eager paging cost-benefit model 53
4.7 Soft page fault tail latency distribution of CBMM compared to other systems 55
4.8 Runtime of workloads on CBMM compared to other kernels 57
4.9 Percent of anonymous memory backed by huge pages on CBMM, HawkEye, and Linux with THP. 58
4.10 Runtime of CBMM workloads with generalized profiles 59
4.11 Soft page fault tail latency distribution for different CBMM profiles 60
4.12 Runtime of CBMM workloads when enabling more models 61
4.13 Soft page fault tail latency impact of different CBMM models 62

5.1 Examples of memory usage patterns. 70
5.2 Amount of memory contiguity for different memory usage patterns 71
5.3 Median % of free memory vs free memory contiguous enough for a 2MB huge page 73

ABSTRACT

Memory management (MM) is a core responsibility of operating system kernels. It has a crucial impact on system performance, efficiency, and cost, especially in large-scale deployments. Meanwhile, computing hardware and software has evolved significantly since the first operating systems were designed. In particular, our work is inspired by two high-level trends: increasing memory sizes and the rise of warehouse-scale computers. The core claim of our work is that system designers must use systematic analysis of MM operations and measurement-based design that prioritizes reasoning about system behavior as a first-class concern. This includes building new tools and methodologies for analyzing system behavior and eliminating heuristic-based policies and *ad hoc* algorithms from MM.

Our work identifies three key problems for modern MM, and takes steps to address them. First, we identify the Capacity Scaling Problem, wherein system software fails to run as well on systems with terabytes of DRAM as it does on current systems. We design, implement, and validate 0sim, a simulator for detecting Capacity Scaling issues in system software using existing commodity hardware. We demonstrate 0sim's utility for debugging and quantifying problems and for prototyping solutions.

Second, we identify sources of kernel MM design and behavioral complexity that hinder performance debugging and improvement, especially at scale. We design, implement, and evaluate Cost-Benefit Memory Management (CBMM), a novel memory management approach based on the idea that all MM operations have a cost and a benefit to userspace and that the benefit should outweigh the cost.

Finally, we quantify the importance of physical memory fragmentation as the need for huge pages and other contiguous memory allocations grows. We conduct a study of physical memory fragmentation in live systems by instrumenting 248 infrastructure and compute cluster machines at UW-Madison's Computer Sciences Department and Center for High-Throughput Computing. We identify memory reclamation and file cache usage as key sources of memory fragmentation on Linux, make several observations about common MM behavior across observed systems, and draw conclusions about several potential improvements to MM.

1 INTRODUCTION

Memory management (MM) is a core responsibility of operating system kernels. It has a crucial impact on system performance, efficiency, and cost, especially in large-scale deployments. Meanwhile, computing hardware and software has evolved significantly since the first modern operating systems were designed. In particular, our work is inspired by two high-level trends: increasing memory sizes and the rise of warehouse-scale computers.

Computer memory sizes have grown continuously since the dawn of computing. Past designs that rigidly limited the maximum amount of memory faced huge difficulties as memory capacity increased exponentially (e.g., IBM's System/360 had an architectural limit of 28MB [15]). This growth trend will continue as technological advances greatly expand the density and decrease the cost of memory. More recently, Intel's Optane memory supported up to 6TB on a two-socket machine [78]. Consequently, multi-terabyte systems may become common, paving the way for future systems with tens to hundreds of terabytes of memory. Increased memory capacities allow more workloads to run but bring challenges too: huge page management becomes more critical due to increased reliance on TLB performance, but memory fragmentation and huge page management overheads also increase with memory capacity [96].

Even as individual machines are getting larger, warehouse-scale computing has made large deployments of machines ubiquitous. In the early 2000s, service providers realized that building bigger, faster, more fault-tolerant servers is an impractical way to handle more traffic. They turned instead to large clusters of bulk-ordered hardware and general-purpose operating systems [20]. At scale, datacenters prioritize *tail* latency as a key service-level metric, in addition to median latency and throughput [53]. Datacenter behavior must be consistent, i.e., low variance, without compromising performance metrics to satisfy service-level objectives and efficiency goals.

In the midst of these hardware and software changes, physical memory fragmentation has emerged as an important but unquantified factor affecting performance. Modern systems require large contiguous physical memory allocations for kernel data structures, caches, suballocators (e.g., slab allocators), and I/O buffers. Moreover, larger memory capacities are placing increasing importance on reducing TLB misses and improving efficiency of memory management mechanisms [96]. A primary means of doing so is the use of huge pages, which require hundreds or thousands of contiguous physical base pages. Fragmentation causes such memory allocations to fail unnecessarily, which can cause system-wide inefficiencies. Surprisingly, little prior work characterizes or quantifies fragmentation, even though many papers refer to its impact and many designs seek to mitigate or avoid it.

In light of these challenges, *the core claim of our work is that system designers must use systematic analysis*

of MM operations and measurement-based design that prioritizes reasoning about system behavior as a first-class concern. This includes building new tools and methodologies for analyzing system behavior and eliminating heuristic-based policies and *ad hoc* algorithms from MM.

1.1 Contributions

Our work is organized along three main thrusts. Each thrust identifies a key problem of current or future system design and seeks to illuminate or mitigate it.

Thrust 1: The Capacity Scaling Problem & 0sim. There is a pressing need to prepare system software for increased memory capacity. While tolerable today, the linear compute and space overheads of many common operating system algorithms may be unacceptable with 10-100x more memory. Other system software, such as language runtimes and garbage collectors also need redesigns to run efficiently on multi-terabyte memory systems [112]. In Linux, many scalability issues are tolerable for small memories but painful at larger scale, as we will show. For example, many current kernel MM algorithms scan through lists of pages, including reclamation, compaction, migration, and huge page promotion, but on systems with hundreds of millions of pages, the cost of scanning becomes onerous.

We expect system designers to encounter new scalability problems as memory sizes grow to terabyte scales and beyond. But, exploring system behavior with huge memories, reproducing scalability issues, and prototyping solutions requires a developer to own or rent a system at great cost or inconvenience. Cloud offerings for 4TB instances cost over $25 per hour [13, 66, 101][1]. Cloud offerings for 11TB or 24TB instances cost over $100 or $200 per hour, respectively, as of this writing; a discounted three-year contract can reduce the cost to $240,000 [13, 101][2].

There is a need for new techniques to understand and prototype system software for huge-memory systems. To this end, we characterize the Capacity Scaling Problem, wherein system software fails to run as well on huge-memory systems as it does on current systems. We discuss several aspects of the Capacity Scaling Problem, including computation inefficiencies, bloated memory usage, impliciations for MM policies, and barriers to development and testing of new systems. In response, we design, implement, and validate the 0sim simulator for detecting Capacity Scaling issues in system software. We demonstrate via case studies that 0sim can be used for debugging and quantifying existing problems and prototyping solutions. For

[1] This was true as of mid-2019, when our original 0sim study was done, and it continues to be true in early 2023, as of this writing.

[2] At the time of the 0sim study in mid-2019, larger (i.e., $> 4\mathrm{TB}$) instances required a three-year contract at an expense of over $780,000 [14, 101]. As of early 2023, some cloud providers allow customers to rent some such machines on demand at premium (e.g., $\geqslant \$100/\mathrm{hr}$), with the possible option of a discounted three-year contract [12, 13, 101]. Other cloud providers still require explicit contracts for 24TB instances [101].

example, we demonstrate how we used 0sim to debug an ironic out-of-memory condition in memcached on large-memory systems, and how we used 0sim to observe large-scale memory fragmentation.

Thrust 2: Kernel MM complexity & Cost-Benefit MM. Current MM designs often fall short of modern computing needs by exhibiting inconsistent, opaque behavior that is difficult to reproduce, decipher, or fix. For example, as we will discuss, current kernel automatic huge page promotion systems often lead to unpredictable performance degradations. These issues come from three key limitations.

1. Kernel MM must predict workload behavior in an information-poor environment. Current MM designs rely on online measurements, particularly page table access/dirty bits and the frequency and location of page faults. Unfortunately, this information is expensive to collect and low bandwidth. For example, access bits can cost up to 11% of CPU cycles to collect [35], but other work finds them insufficient to predict TLB miss overheads accurately [113].

2. Current MM designs often ignore the cost of various MM operations, leading to inappropriate policy decisions. For example, Linux allocates a huge page when a memory region is first touched; however, we find that allocating and zeroing a huge page costs a million cycles ($\sim 500\mu sec$ on a 2GHz processor) in the best case. Thus, promoting a page that averts less than a million cycles worth of TLB misses and page faults actually *regresses* performance, but the kernel does not account for this cost.

3. Current MM designs are implemented as disjointly acting policies distributed throughout the kernel that are hard to debug. This opaque system implementation and its consequent opaque behavior is a primary obstacle to improving kernel MM performance, consistency, and debuggability, and it often leads to suboptimal fixes such as disabling huge pages altogether [5,49,102,111,143].

In our work, we demonstrate an approach to MM based on thorough measurement of system behavior to inform system design and policies. We perform a detailed study of the sources of cost and benefit in soft page fault handling and huge page management. We find that in many cases, kernel heuristics opt to do expensive operations, such as zeroing huge pages, in cases where they are not worth their cost. To mitigate this problem we propose and prototype Cost-Benefit Memory Management (CBMM), a MM system based on our measurements and designed to prioritize reasoning about system behavior. CBMM is based on the idea that all MM operations have a cost and a benefit to userspace applications, and the benefit should outweigh the cost. We evaluate CBMM via micro- and macro-benchmarks to demonstrate the potential of our approach.

Thrust 3: Physical memory fragmentation. Fragmentation causes memory allocations to fail unnecessarily. *External* fragmentation occurs when free memory is too discontiguous to satisfy requests: when adjacent differently sized allocations have differing lifetimes, free memory will interleave with allocated memory. (It is distinct from *internal memory fragmentation*, where an allocated block of memory is underutilized). Both userspace and kernel memory managers suffer fragmentation, but we focus on *kernel* memory management for *physical memory*.

In current systems, physical memory fragmentation can affect transparent huge page procedures strongly by forcing them to defragment memory [115]. On Linux, we found that such defragmentation can degrade page fault latency by 50% to 500%, as we will show in Chapter 4. Moreover, as we will show in Chapter 5, other important uses for contiguous memory are hindered by fragmentation, such kernel data structures and I/O buffers.

Surprisingly little prior work exists to characterize or quantify physical memory fragmentation, even though many papers refer to its impact and many designs seek to mitigate or avoid it. For example, it is unclear what contiguous memory allocations are actually used for, the degree to which workloads fragment system memory, what types of memory allocations cause fragmentation, how fragmentation changes over time, etc.

For this reason, we conduct a study of physical memory fragmentation on modern systems. We instrument 248 live systems in the CS department at UW-Madison, including departmental infrastructure and a high-performance computing cluster. We analyze the amount, usage, and contiguity of free and allocated physical memory in the system over 1 week. Our findings show that Linux uses contiguous memory mostly for huge pages (512 contiguous pages), buffers, and kernel slabs for data structures (2-8 contiguous pages). Moreover, we identify several common memory usage patterns and find that the file cache and reclamation algorithm are key sources of fragmentation.

1.2 Organization of this Dissertation

Chapter 2 discusses related work and background information about the current state of kernel MM. Chapters 3, 4, and 5 discuss the three thrusts described in the previous section. Finally, we share observations, experiences, and conclusions in Chapters 6 and 7.

2 BACKGROUND

Memory Management (MM) is a broad area of systems programming that concerns the multiplexing and mapping of physical and virtual memory. Significant prior work exists to create userspace allocators, language runtimes, static analyses, and libraries to make userspace memory management faster, more efficient, and more cost-effective; however, we limit our scope to OS kernel MM, rather than userspace. In current mainstream OSes, the kernel MM is tasked with allocating/reclaiming and mapping/unmapping physical memory into the virtual address space of different processes. In this chapter, we give an overview of the state of kernel MM in current mainstream systems software, focusing on Linux (because it is open source and widely used) and on server-class machines (because our work focuses on the datacenter setting), along with the challenges that motivate our work.

2.1 Virtual Memory

Excluding embedded systems, all wide-spread modern architectures (e.g., x86/x86_64 and ARM) today use virtual memory and paging: the kernel and processor collaborate to create an indirection layer between the memory addresses programs use and those used by memory hardware. The indirection layer is managed via a collection of tables (called *page tables*) that translate virtual addresses (those used by programs) into physical addresses (the actual hardware location). The granularity of this translation is generally several kilobytes at a time (e.g., 4KB on x86/x86_64) and is called a *page*.

Virtual memory can implement security isolation, memory overcommitment, simpler programming models, communication mechanisms, and more. Modern OSes use virtual memory in several ways:

Anonymous memory is memory mapped into the address space of a process but not backed by data in a file system. Usually, most anonymous memory is used for a process's stack and heap, but it can also be used for static/global variables, memory-mapped I/O, or interprocess communication buffers/shared memory. Linux keeps track of the working set of a process using a hybrid LRU/clock-like algorithm, as discussed later in this Section. Under memory pressure, anonymous pages may be selected to be written to disk (swap space) and then freed for other usage.

File/buffer cache memory is memory used for buffering/caching contents backed by a file system [123]. As with anonymous memory, Linux reclaims file cache memory in LRU/clock-like order. A page can be reclaimed quickly if it is "clean", i.e., the memory contents match the disk contents; otherwise, the page is "dirty" and needs to be flushed to disk before being freed.

Kernel memory is memory used by the kernel itself. The kernel uses memory for its own stack and data structures, many of which are long-lived. In Linux, kernel pages are always pinned in memory – they cannot be moved or reclaimed until the kernel explicitly frees them. Some kernels, such as Windows, allow some kernel pages to be reclaimed or moved. Kernel pages are usually inaccessible from userspace.

Driver/Module memory is memory requested by a driver or kernel module for some purpose, often for a buffer or queue. In Linux, such memory is opaque to the kernel and cannot be moved or reclaimed unless explicitly allowed by the driver via a registered callback function. Like kernel pages, driver/module memory is usually inaccessible from userspace.

2.2 What does Memory Management do?

It is the kernel MM's job to track and multiplex many virtual address spaces onto limited physical address space. The policies and mechanisms used to do so greatly affect system behavior and performance. In our work, we argue that many of these policies and mechanisms need to be redesigned in light of the modern datacenter environment.

Memory Allocation and Reclamation. Virtual memory regions are tracked and allocated per-process, since each process has a separate address space (in current mainstream OSes, at least). Processes request virtual memory allocations through a system call such as `mmap`. Allocating a virtual memory region usually does not allocate any physical memory; rather, the region is marked as accessible. Upon touching the virtual memory region, the process will incur a page fault, and physical memory will then be allocated. This scheme for lazily allocating memory is called *demand paging*, and it reduces memory bloat from requested-but-unused memory at the cost of extra page faults. In Linux, prior to version 6.1, allocated virtual memory regions are tracked via a red-black tree of `vm_area_struct` structs. Going forward, they will be tracked with maple trees [126], which simplify some locking and programming problems.

In contrast, phyiscal memory is limited by the hardware physically installed in the system. In Linux, each page of physical memory is tracked via a `struct page` – other kernels have similar data structures. `struct page` comprises a number of bit flags, locks, and pointers to other data structures, whose purpose changes with context; for example, the struct marks the position of pages in LRU lists or whether a page is currently being used for file I/O. Moreover, on NUMA machines, Linux keeps separate lists of `struct page` for each NUMA node.

Generally, there is single global kernel physical memory allocator that feeds all other memory allocation mechanisms. Because of the frequency of physical memory allocations from both userspace and the kernel

itself, the allocator needs to be fast. On the other hand, to increase memory usage efficiency, the allocator needs to keep minimal metadata and should minimize internal memory fragmentation. Linux and FreeBSD both use a *buddy allocator*. Buddy allocators satisfy all requests with power-of-two-sized-and-aligned blocks (e.g., a thread requesting 3 pages is allocated 4 pages; a thread requesting 257 pages is allocated 512 pages. Because of the size and alignment requirements, the allocator can always tell the boundaries of an allocation when it is freed.). The allocator maintains free lists for all power-of-two-sized blocks up to some maximum size. If a block of size N is needed but none is available, a block of size (and alignment) 2N is acquired and split into two "buddy" blocks – one is used to satisfy the allocation, and one is placed on the N-sized free list. When a page is freed, it is trivial to find the page's "buddy" because of the alignment requirements; if the buddy is free, the two blocks can be merged back into a 2N-sized-and-aligned block and placed on the 2N-sized free list. Generally, Linux allocates the head element on the appropriate free list. However, some minimal effort is made to allocate pages that are likely still hot in the caches, such as placing cold or contended pages on the tail of the list.

Under memory pressure (i.e., when free memory is scarce), the kernel must either fail new allocations or take memory away from current users. The process of taking memory from current users is *reclamation*. On Linux and FreeBSD, the selection of which pages to take away is done via a LRU-like or clock-like [36] algorithm (or some hybrid thereof). While the exact mechanism is under evolution, the general principle is to have multiple levels of LRU-ish lists; pages that are cold/idle/unused eventually end up in the lowest level and are eventually selected to be reclaimed – possibly after being flushed to disk. The coldness/idleness of a page is determined by page accessed/dirty bits in the page tables; the bits are set by the processor when the page is accessed/written to, and the kernel can scan page tables for these bits to check if the page has been used. In Linux, each NUMA node and cgroup (used to implement containers) has a set of LRU lists – a hot and cold list for anonymous memory, and another pair for file cache memory. The separation of such lists allows reclamation to be targetted at a particular NUMA node or container.

Mapping and Unmapping. Before physical memory can be used by a process (or the kernel), it must be mapped into a virtual address space. There are a large array of choices of *how* to map a page. Alongside the address translation, page table entries also contain a variety of bits that control the type of mapping. Some bits indicating whether a page is read-only, writable, executable, userspace-accessible, mapped in all address spaces, etc. Other bits indicate that a page is a huge page, uncachable, etc.

By using different page table configurations and sets of bits, the kernel can achieve different effects. Copy-on-write is achieved by marking a given page as read-only; attempts to write the page will cause a page fault, and the kernel can then copy the page contents and map the new and old copies as writable. Shared

memory can be implemented by mapping the same physical page from multiple page table entries (e.g., in different address spaces).

Memory is unmapped by clearing the "present" bit from the relevant page table entries. Modern processors cache translations in a hardware cache called the Translation Lookaside Buffer (TLB), so the kernel must flush the relevant entries of the TLB too (often this requires interrupting other processor cores to inform them that they should flush their TLBs).

Fragmentation Control. *Memory fragmentation* is a system state in which memory allocations fail despite having enough unused memory. *External memory fragmentation* happens when free memory is too discontiguous to satisfy allocations, even though the total amount of free memory is larger than the requested amount. *Internal memory fragmentation* is underutilization of an allocated block of memory.

Generally, because kernels control their own allocations and data structure sizes, internal fragmentation is less of a problem (e.g., Linux uses slab allocators that try to pack similar data structures into allocated memory). Likewise, because virtual address spaces are not (practically) limited and each address space is quite large, virtual memory fragmentation is less of a concern (at the kernel level; userspace allocators are quite concerned about it). Thus, throughout the rest of this dissertation, when we refer to memory fragmentation, we mean external fragmentation of physical memory.

Fragmentation is significant because it hinders the allocation of large contiguous chunks of physical memory. As we will see in Chapter 5, such contiguous memory is needed for kernel data structures, buffers, and huge pages. On a fragmented system, these allocations may fail or be very slow. Thus, Linux and FreeBSD contain fragmentation control mechanisms.

One kind of fragmentation control mechanism operates at allocation time by trying to preserve contiguity. Buddy allocators do this naturally: merging free buddy blocks keeps large contiguous regions separate from less contiguous regions; having separate free lists for different sized blocks allows smaller allocations to be satisfied from less contiguous regions, leaving larger regions intact longer [118]. FreeBSD also features a *reservation* system that defers the allocation of potential huge pages as long as possible [105]. These mechanisms are low-overhead and passively reduce fragmentation, but can have limited effectiveness.

In contrast, Linux also features a "compaction" (defragmentation) daemon that actively migrates movable memory contents away from otherwise contiguous memory. This process can be expensive and can harm system performance by hogging CPU time, disrupting caches, and locking memory in use by processes while it is migrated (more later) [87,115].

2.3 Challenges in Modern Memory Management

Our work adds to significant prior work from both industry and academia that adapts MM to modern needs. In this section, we outline several challenges that current MM designs are struggling to address.

Huge Pages

The increasing disparity between processor speeds and memory speeds, necessitates address translation caches: Translation Lookaside Buffers (TLBs). TLBs are on the critical path of instruction execution, so they must be fast and highly-associative structures, making them expensive in terms of chip area and energy. Current generations of hardware are typically limited to about 2000 entries, which is enough to cache translations for $2000 \times 4\text{KB} = 8\text{MB}$ of memory [142,144]. 8MB is $< 0.1\%$ of memory on an inexpensive laptop, let alone a modern server; 8MB is smaller than the working set of many server applications. Applications with large working sets or random memory accesses suffer from high TLB miss rates and consequent processor stalls that can consume over 50% of execution time in some cases [21,83]. As a solution, hardware vendors have implemented larger page sizes (often called 'huge pages' or 'large pages' or 'superpages'; 2MB and 1GB on x86_64) so that the same number of TLB entries covers a larger portion of memory.

However, systems and applications struggle to use huge pages well. Windows, Linux, and BSD all have system calls for applications to explicitly request huge pages [11,87,144], but (1) they require changing application/library source code to request and benefit from huge pages and (2) application developers often do not know how/whether their application would benefit from huge pages [87]. Thus, Linux and BSD attempt to transparently and automatically "promote" memory regions to be backed by huge pages without applications requesting it, obviating application changes or developer expertise.

Unfortunately, current transparent huge page (THP) implementations have significant problems. BSD's THP feature (based on Navarro et al's work [105]) is too conservative. Its "reservation"-based system waits for applications to touch a majority of the base pages in a potential huge page region before it promotes the region to a huge page. This avoids bloating memory usage if an application does not need the whole huge page, but it also allows the application to experience many TLB misses before huge page promotion, leaving performance on the table [87]. In contrast, Linux's THP is overly aggressive: it promotes memory regions immediately the first time they are touched. This allows applications to get the maximum benefit from huge pages, but it can bloat memory usage. Worse, the increased demand for properly aligned contiguous memory often triggers active memory defragmentation whose overhead can degrade application performance unpredictably, leading many applications to recommend turning the feature off [5,49,87,102,111,143].

Recent work attempts to address these challenges. Ingens, HawkEye, and Translation Ranger improve

Linux with heuristics for selecting which pages to promote, how and when to compact memory, and how to balance huge page usage fairly between processes [87,113,142]. Quicksilver attempts the same but based on BSD's implementation, rather than Linux [144]. Panwar et al find that kernel pinned pages can significantly degrade the efficacy of memory compaction and propose changes to page placement and compaction [115]. None of these approaches have been adopted into a mainline kernel in Linux or BSD yet, and more research is needed. In particular, as we will show in Chapter 4, current designs suffer from behavioral complexity that makes implementation and performance debugging difficult.

Fragmentation

Memory fragmentation is an old problem [67,68], but it has recently come to the forefront again in recent work because it hinders the use of huge pages [115,144]. For example, we found that fragmentation degrades application performance on Linux by 50% to 500% due to failed or delayed attempts to create huge pages.

Prior work extensively studies usermode allocators, including internal and external fragmentation control, use of huge pages, and more [60,75,80,120,124]. However, surprisingly little prior work characterizes physical memory fragmentation from the perspective of kernel MM, despite the fact that many papers acknowledge its impact. Panwar et al discuss the importance of kernel pinned page placement [115], and Zhu et al suggest that file cache pages play an important role in fragmentation [144], but no prior work quantifies or characterizes in detail what fragmentation looks like in the wild. In Chapter 5, we fill this gap in knowledge.

Scaling with Memory Capacity

Extensive prior work has examined different kinds of scalability problems in system software. For example, RadixVM tries to overcome performance issues in highly concurrent workloads due to serialization of memory management operations on kernel data structures [38]. Other studies have suggested that `struct page`, `struct vm_area_struct`, and page tables tend to comprise a large portion of memory management overhead [59]. Java 11 features a new garbage collector that allows it to scale to terabyte-scale heaps while retaining low latency [112]. However, overall, there has been fairly little work aimed at improving the scalability of systems with respect to the memory capacity.

While tolerable today, the linear compute and space overheads of many common operating system algorithms may be unacceptable with 10-100x more memory. Other system software, such as language runtimes and garbage collectors also need redesigns to run efficiently on multi-terabyte memory systems [112]. In Linux, many scalability issues are tolerable for small memories but painful at larger scale. For example:

- With 4TB of memory, Linux takes more than 30s at boot time to initialize page metadata, which reduces availability when restarts are needed.
- Kernel metadata grows to multiple gigabytes on large systems. On heterogeneous systems, metadata may overwhelm smaller fast memories [41].
- Memory management algorithms with linear complexity, such as memory reclamation and defragmentation, can cause significant performance overhead when memory scales up 10x in size, as we show later.
- Huge pages are critical to TLB performance on huge systems, but memory defragmentation to form huge pages has previously been found to cause large latency spikes for some applications [5,49,102,111].
- Memory management policies built for smaller memories, such as allowing a fixed percentage of cached file contents to be dirty, perform poorly with huge memories when that small percentage comprises hundreds of gigabytes [103].

We expect system designers to encounter new scalability problems as memory sizes grow to terabyte scales and beyond. But, exploring system behavior with huge memories, reproducing scalability issues, and prototyping solutions requires a developer to own or rent a system at great cost or inconvenience. Thus, there is a need for new techniques to understand and prototype system software for huge-memory systems. In Chapter 3, we propose 0sim, a simulator designed to help understand and prototype solutions for these problems.

Changes in Memory Architectures

Technologic and economic trends have combined to incentivize a wide variety of new memory architectures and MM designs. Today, servers commonly have multiple sockets, each with a memory controller and attached DRAM, leading to Non-Uniform Memory Access (NUMA) architectures, in which a core must endure significantly longer memory access latency for memory attached to a remote socket [90]. New memory technologies such as persistent memory or Compute eXpress Link (CXL) have made memory tiering and disaggregation more cost effective and practical [92,99]. While our work does not directly address NUMA or heterogeous memory technologies, our MM techniques, tools, and findings can help prepare MM for a world where these technologies are ubiquitous. In Chapter 3, we identify scalability challenges for large-memory systems and propose tools to study them. And in Chapter 4, we discuss how to make MM behavior more predictable and debuggable, a prerequisite for the more complex MM policies needed to support new memory architectures.

Policy	Goal	Pathology
Huge Page Allocation	Reduce TLB misses and page faults	Bloat memory usage if not all memory is used; increase page fault latency if compaction is required
Copy-on-Write or Demand Paging	Reduce unnecessary memory bloat and initialization time	Increase page fault latency post-initialization if the page is written/accessed
Eager Paging [83]	Move page fault latency to allocation time, saving time later	Bloat memory usage if not all memory is used
Background Compaction	Reduce memory fragmentation and huge page fault latency	Increase CPU overhead
Background Zeroing	Reduce page fault latency	Increase CPU overhead
Idle Page Reclamation [88, 138]	Improve memory utilization	Increase overhead to fault reclaimed pages back in; increase CPU overhead to choose pages to reclaim

Table 2.1: Different MM policies and their goals and pathologies

Meanwhile, the end of Denard Scaling, exponentially increasing amounts of data, and stagnating DRAM costs incentivize service providers to improve Total Cost of Ownership (TCO) by migrating data between heterogenous memory tiers and types. Facebook and Google both implement "far-memory" migration systems, in which cold memory contents are moved to slower media to make room for hot contents and improve hardware utilization [88,140]. Facebook servers use locally-derived Pressure Stall Information to guide offloading to compressed memory or an SSD [140]. Google makes use of centralized planning via aggregated page table access bits to guide memory compression (via Linux's zswap system) to all machines in a cluster [88]. At the same time, efforts in the Linux community seek to improve the design of memory reclamation (e.g., by having multiple "generations" of LRU lists beyond the two existing ones) due to the increasing prevalence of super-fast storage devices which can make swapping a viable means of increasing machine capacity [47]. As with new hardware memory architectures, our work provides tools and techniques to support these new MM paradigms and use cases, for example by making MM policies more predictable and, thus, more suitable for large-scale applications (Chapter 4).

MM Behavioral Complexity and Visibility

A recurring theme in our work is a pressing need for more and better visibility into system behavior, both at runtime for guiding MM and offline for system development and debugging. Our work proposes new tools and new methodologies for understanding system behavior (Chapters 3 and 4), new systems that are easier to understand (Chapter 4), and new data about the behavior of existing systems (Chapter 5).

MM mechanisms and policies comprise a complex set of tradeoffs that, when poorly navigated, lead to poor performance or unexpected behavior. Meanwhile, datacenters prioritize *tail* latency as a key service-level metric, in addition to median latency and throughput [53]. Datacenter behavior must be consistent, i.e., low variance, without compromising performance metrics to satisfy service-level objectives and efficiency goals.

Performance consistency at scale is a well-known problem [53] afflicting, among other systems, cluster

computations [54] and distributed caching [26]. Redundancy is a common workaround [54]. MittOS uses deadline-aware kernel APIs to improve tail latency [71]. Kwon et al. observe that current huge page support is "a hodge-podge of best-effort algorithms and spot fixes" [87]. As previously discussed, they and others identify real concerns and improve performance but often at the expense of increasing kernel heuristic complexity [32, 87, 113, 115, 142].

Unfortunately, current MM designs still fall short of modern computing needs by exhibiting inconsistent, opaque behavior that is difficult to reproduce, decipher, or fix. For example, we found that for some workloads on Linux, a soft page fault lasting 25ms occurs every 100ms. This drastic tail latency is due to memory compaction or reclamation when attempting to allocate a huge page – a misnavigated tradeoff. Many applications would violate response latency objectives if one request per 100ms takes 25ms due to a page fault. As a result, Redis, MongoDB, and others advise users to disable Linux's Transparent Huge Page (THP) feature [5, 49, 102, 111, 143]. Table 2.1 lists other examples of MM policies and their potential pathologies. In Chapter 4, we identify three key limitations that cause these issues and propose solutions.

Visibility into system behavior is also key to understanding the Capacity Scaling Problem; a key challenge of our work is the unavailability of hardware. Simulation is often used when hardware is unavailable (e.g., processor simulators [31]). Unlike other hardware advances, such as increasing processor cores or device speed, simulating memory capacity on existing hardware is challenging due to the amount of state that must be maintained. Alameldeen et al. address this by painstakingly scaling down and tuning the benchmarks and systems they test [10]. While accurate, this methodology is error prone, tedious, and difficult to validate. In Quartz and Simics, simulation size is limited by the size of the host machine [59, 137]. Researchers have simulated fast networks by slowing down the simulated machine's clock [70]. David [8] and Exalt [139] simulate large storage systems by storing only metadata and generating contents at read time. This technique is difficult for memory because important metadata is not separated from disposable data in most programs. Several systems virtualize a cluster of machines to produce a single large virtual machine [23, 37, 127, 133]; Firesim uses cloud-based accelerators to scale simulations [84]. In Chapter 3, we describe 0sim, a simulator for studying the Capacity Scaling Problem, which priortizes capacity and speed at a modest cost in accuracy.

2.4 Target Setting and Scope

Our work targets kernel MM for first-party datacenter workloads – workloads that service providers run on their own infrastructure – a critical piece of modern cloud computing.

Modern large-scale applications are commonly deployed on warehouse-scale computers (WSCs). In a WSC, large numbers of commodity servers are used with distributed algorithms to give the illusion of a

single high-capacity, high-availability server. By nature, WSCs operate at large scales, often with tens or hundreds of thousands of servers and dozens or hundreds of interdependent services/applications [20].

Several interesting properties emerge from this environment:

- Tail-latency is key service-level performance metric. Incoming user requests frequently have high fanout to backend-services, so the latency of the overall response is tied to the latency of the slowest dependee service [53].
- First-party hardware and software are highly redundant. Generally, hardware is procured and deployed in bulk. Meanwhile, identical redundant copies of services are on many machines to increase fault-tolerance and capacity [20]. As a result, there is a level of homogeneity among hardware and software in a cluster – very few machines are unique and many are identical. Moreover, many workloads run continuously for long periods of time [20,28,65,79,100,129,132,134].
- First-party hardware and software are highly controlled. Usually, only the WSC operator has physical access to deploy hardware. Meanwhile, software teams often deploy their first-party services along a well known release schedule, leading to relatively incremental changes.
- Disparate workloads are packed onto the same machines. In order to increase system utilization (to improve efficiency), service providers pack tasks onto machines, leading to highly concurrent and highly diverse workloads [20,28,65,79,100,129,132,134].

3 0sim: preparing system software for a world with terabyte-scale memories

A whole new world (Don't you dare close your eyes)
A hundred thousand things to see (Hold your breath, it gets better)
I'm like a shooting star, I've come so far
I can't go back to where I used to be (A whole new world)

With new horizons to pursue
I'll chase them anywhere
There's time to spare
Let me share this whole new world with you

— "A Whole New World", Aladdin, Disney, 1992

Acknowledgements. *The contents of this chapter were published in ASPLOS 2020 and are co-authored with Michael M. Swift [96].*

We would like to thank Mark Hill, Swapnil Haria, Ram Alagappan, Yuvraj Patel, Anjali, and the anonymous reviewers for their insightful and helpful feedback. We would like to thank Bijan Tabatabai, Suhas Pai, Hasnain Ali Pirzada, and Varun Ravipati for their help in exploring workloads and developing infrastructure at various points in this project. We would like to thank GitHub user and memcached maintainer `dormando`*, for their aid in debugging the memcached issues mentioned in Section 3.6. This works is supported by the National Science Foundation under grant CNS-1815656.*

Computer memory sizes have grown continuously since the dawn of computing. Past designs that rigidly limited the maximum amount of memory faced huge difficulties as memory capacity increased exponentially (e.g., IBM's System/360 had an architectural limit of 28MB [15]). This growth trend will continue as technological advances greatly expand the density and decrease the cost of memory. Most recently, Intel's 3D Xpoint memory supports up to 6TB on a two-socket machine [78]. Consequently, multi-terabyte systems may become common, paving the way for future systems with tens to hundreds of terabytes of memory.

There is a pressing need to study the scalability of system software and applications on huge-memory systems (multiple TBs or more) to prepare for increased memory capacity.

To this end, we built and open-sourced 0sim ("zero·sim"), a virtualization-based platform for simulating system software behavior on multi-terabyte machines. 0sim runs on a commodity host machine, the *platform*, and provides a user with a virtual machine, the simulation *target*, which has a huge physical memory. 0sim is fast enough to allow huge, long-running simulations and even direct interaction, enabling both performance measurements and interactive debugging of scalability issues.

We employ several novel techniques to fit terabytes of target memory contents into gigabytes of platform memory. First, we observe that many programs are *data oblivious*: they perform the same computation independent of input values. Therefore, we can exercise the target with predetermined input; then, 0sim can compress a 4KB page with predetermined content to 1 bit. Second, current processors may have small physical address spaces to simplify the CPU design. We use software virtualization techniques to ensure that simulated physical addresses are never seen by the platform CPU, but instead are translated by 0sim. Thus, our system can simulate the maximum allowable address space for a given target architecture. Finally, we enable efficient performance measurement within the simulation by exposing hardware timestamp counters that report the passage of target time.

0sim simulates both functional and performance aspects of the target as if measured on a real multi-terabyte machine. 0sim does not aim to be an architecture simulator or to be perfectly accurate, as the target may differ from the platform in many ways including processor microarchitecture. Instead, 0sim simulates system software *well enough* for the important use cases of reproducing scalability problems, prototyping solutions, and exploring system behavior. 0sim sacrifices some accuracy for simulation speed, enabling huge, long-running simulations and interactive debugging.

In this paper, we describe the architecture and implementation of 0sim. We validate 0sim's accuracy and simulation speed with these goals in mind. 0sim can simulate a target system 20-30x larger than the platform with only 8x-100x slowdown compared to native execution for the workloads we tested, with more compressible workloads running faster. For example, we simulate a 4TB memcached target on a 160GB platform and 1TB memcached target on a 30GB platform with only 8x slowdown. By comparison, architecture simulators incur 10,000x or worse slowdown [31].

We perform several case studies demonstrating the usefulness of 0sim: we reproduce and extend developer performance results for a proposed kernel patch; we measure the worst-case impact of memory compaction on tail latency for memcached as a 22x slowdown; we show that for a mix of workloads Linux can incur irreparable memory fragmentation *even with dozens of GBs of free memory*; and that synchronous page reclamation can be made much more efficient at the cost of very small additional delay. Furthermore, we used 0sim to interactively debug a scalability bug in memcached that only occurs with more than 2TB of memory. 0sim is available at `https://github.com/multifacet/0sim-workspace`.

3.1 The Capacity Scaling Problem

0sim addresses a critical need to study how software scales with memory capacity, including making efficient use of memory, addressing algorithmic bottlenecks, and designing policies with huge memory capacity in mind. Moreover, it removes barriers for developers that limit software from being effectively tested and deployed for huge-memory systems.

Computational Inefficiency. Any operation whose execution time increases linearly with the amount of memory may become a bottleneck. For example, the Page Frame Reclamation Algorithm, huge page compaction, page deduplication, memory allocation, and dirty/referenced bit sampling all operate over per-page metadata. If the kernel attempts to transparently upgrade a range of pages to a huge page, running huge page compaction may induce unpredictable latency spikes and long tail latencies in applications. This has led many databases and storage systems to recommend turning off such kernel features despite potential performance gains [5,49,102,111].

Likewise, allocating, initializing, and destroying page tables can be expensive. This impacts the time to create and destroy processes or service page faults. Linus Torvalds has suggested that the overhead of allocating pages to pre-populate page tables makes the `MAP_POPULATE` flag for `mmap` less useful because its latency is unacceptably high [136].

Another example is the Linux kernel's `struct page` [42]. Huge-memory systems may have billions of these structures storing metadata for each 4KB page of physical memory. In a 4TB system, initializing each of them and freeing them to the kernel's memory allocator at boot time takes 18 and 15 seconds, respectively, on our test machines. This has implications for service availability, where the boot time of the kernel may be on the critical path of a service restart.

Memory Usage. Any memory usage that is proportional to main memory size may consume too much space in some circumstances. For example, a machine with a modest amount of DRAM and terabytes of non-volatile memory may find all of its DRAM consumed by page tables and memory management metadata for non-volatile memory [41].

As previously mentioned, for each 4KB page, the Linux kernel keeps a 200-byte `struct page` with metadata. Similarly, page tables for address translation can consume dozens of gigabytes of memory on huge systems [59]. While this space may be a small fraction of total memory, it consumes a valuable system resource, and as discussed above, imposes a time cost for management.

Huge-Memory-Aware Policies. Effective memory management policies for small memories may perform poorly at huge scales. For example, the Linux kernel used to flush dirty pages to storage once they exceeded a percentage of memory, but on huge machines this led to long pauses as gigabytes of data were flushed out [103]. Also, some applications that use huge memories to buffer streaming data have found the kernel page cache to be a bottleneck [45]. In an era of huge and/or non-volatile memories, its not clear what role page caching should play. As high-throughput, low-latency network and storage and huge memories become more common, it is important to reevaluate kernel policies for buffering and flushing dirty data.

Likewise, policies for large contiguous allocations and fragmentation control need to be examined. Many modern high-performance I/O devices, such as network cards and solid-state drives, use large physically contiguous pinned memory buffers [50]. Researchers have proposed large contiguous allocations to mitigate TLB miss overheads [21,63]. In order to satisfy such memory allocations, the kernel must control fragmentation of physical memory, which has been a problem in Linux [43,45]. A complementary issue is the impact of internal fragmentation caused by eager paging [63] and transparent huge pages on multi-terabyte systems. These problems are well-studied for smaller systems, but to our knowledge they have not been revisited on huge-memory systems and workloads.

Capacity scalability problems are not unique to operating systems. Other system software, such as language runtimes, also needs to be adapted for huge memory systems. For example, Oracle's Java Virtual Machine adopted a new garbage collector optimized for huge-memory systems [112].

Barriers to Development. Huge-memory systems are uncommon due to their expense. Hence, system software is not well-tested for huge-memory systems that will become common soon. In our work, we often found that software had bugs and arbitrary hard-coded limits that caused it to fail on huge-memory systems. Usually, limitations were not well-documented, and failures were hard to debug due to user-unfriendly failure modes such as unrelated kernel panics or hanging indefinitely. `0sim` makes it easier for developers to test software at scale.

3.2 Design of `0sim`

`0sim` enables evaluation of *system software* on machines with huge physical memories. We emphasize that `0sim` is not an architecture simulator; instead it has the following goals:

- Run on inexpensive commodity hardware.
- Require minimal changes to simulated software.

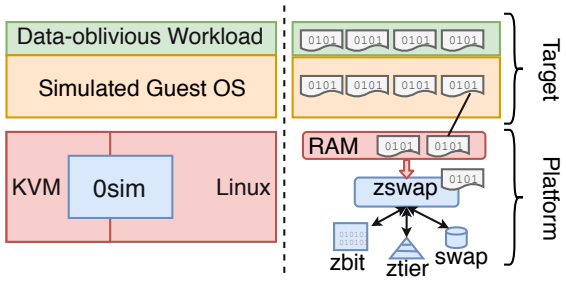


Figure 3.1: Design of 0sim (left) and Simulation State (right).

- Preserve performance trends, not exact performance.
- Run fast enough to simulate long-running workloads.

Figure 3.1 (left) shows an overview of 0sim's architecture. 0sim boots a virtual machine (VM), the *target*, with physical memory that is orders-of-magnitude larger than available physical memory on the host, or *platform*, while maintaining reasonable simulation speed. 0sim is implemented as a modified kernel and hypervisor running on the platform but requires no target changes. *Any unmodified target OS* and a wide variety of workloads can be simulated by executing them in the target (e.g., via SSH). The x86 `rdtsc` instruction can be used in the target to read the hardware timestamp counter (TSC) for simulated time measurement.

0sim trades off simulation speed and ease of use of the system against accuracy by seeking to preserve trends rather than precisely predict performance. 0sim preserves trends in both temporal metrics (e.g., latency) and non-temporal metrics (e.g., memory usage) in the simulated environment. For example, 0sim can be used to compare the performance of two targets to measure the impact of an optimization.

The central challenges facing 0sim are (1) emulating huge memories and (2) preserving temporal metrics. We address (1) using data-oblivious workloads and memory compression. We address (2) by virtualizing the TSC.

Techniques. 0sim's design and implementation make use of a number of known software techniques to efficiently overcommit memory. One of our contributions is to show how 0sim uses these techniques to build a novel approach to simulation. Page compression and deduplication can increase memory utilization in the presence of overcommitment; they are implemented in widely-used software, such as Linux, MacOS, and VMware ESX Server [3,4,16,138]. In Linux, work has been done to increase the achievable memory compression ratio by using more efficient allocators [104] and optimizing for same-filled pages [55]. Remote-memory proposals swap pages out over the network to a remote machine, allowing larger workloads to run

locally [61, 69]. Work has also been done on hardware-based memory compression [7] and zero-aware optimizations [58], but these proposals require specialized hardware, unlike 0sim.

Data-Obliviousness

Simulating huge-memory systems is fundamentally different from simulating faster hardware, such as CPUs or network devices. As previously mentioned, simulating huge memories requires maintaining more state than the platform is capable of holding. 0sim relies on the platform kernel's swapping subsystem to transparently overflow target state to a swap device. However, the platform may not have enough swap space for the state we wish to simulate; even if it did, writing and reading all state from the storage would be painfully slow and would make huge, long-running simulations impractical.

Our key observation is that many workloads follow the same control flow regardless of their input. We call such workloads *data-oblivious*. For example, the memcached in-memory key-value store does not behave differently based on the *values* in key-value pairs – only the keys. Another example is fixed computation, such as matrix multiplication; we can provide matrix workloads with sparse or known matrices. One workload, the NAS Conjugate Gradient Benchmark [19], naturally uses sparse matrices.

Figure 3.1 (right) depicts the management of target state in 0sim. Providing predetermined datasets to a data-oblivious workload makes it highly amenable to memory compression without changing its behavior. 0sim recognizes pages with the predetermined content (e.g., a zeroed page) and compresses them down to 1 bit, storing them in a bitmap called *zbit*. Pages that do not match the predetermined content can instead be compressed and stored in a highly efficient memory pool called *ztier*. This allows 0sim to run huge workloads while maintaining simulation state on a much more modest platform machine. Moreover, because much of the simulation state is kept in memory, zbit enables much faster simulation than if all state had to be written to a swap device. For example, on our workstations writing 4KB to an SSD takes about 24µs, while LZO compression [109] takes only 4µs.

0sim depends on data-obliviousness for simulation performance. Some interesting workloads are difficult to make data-oblivious, such as graphs and workloads with feedback loops. Nonetheless, to study system software, such as kernels, data-oblivious workloads usefully exercise the system in different ways including different memory allocation and access patterns. Thus, we believe data-oblivious workloads are sufficient to expose numerous problems, and that many of our findings generalize to other workloads. For example, much of the kernel memory management subsystem can be exercised because it is agnostic to page contents. We demonstrate this using several case studies in Section 3.6. Moreover, preparing systems for data-oblivious workloads benefits non-data-oblivious workloads too.

Hardware Limitations

Existing commodity systems may not support the amounts of memory we wish to study. For example, one of our experimental platforms has 39 physical address bits, only enough to address 512GB of memory, whereas we want to simulate multi-terabyte systems. This hardware limitation prevents running huge-memory workloads. 0sim overcomes the address-size limitation using shadow page tables [33]: the hypervisor, not hardware, translates the target physical addresses to platform physical addresses of the appropriate width. While not implemented, targets running virtual machines [24] or with 5-level paging, which was announced by Intel but is not yet widely supported [76], can also be simulated with this technique. Similarly, 0sim supports memory sizes larger than the available swap space by transparently using memory compression in the hypervisor to take advantage of workload data-obliviousness.

Time Virtualization

Hardware simulators, such as gem5 [31], often simulate the passage of time using discrete events generated by the simulator. However, this is extremely slow, leading to many orders-of-magnitude slowdown compared to native execution. Such slowdowns make it impractical to study the behavior of huge-memory systems over medium or long time scales. Instead, 0sim uses hardware timestamp counters (TSCs) on the platform to measure the passage of time.

Each physical core has an independent hardware TSC that runs continuously. However, there are numerous sources of overhead in the hypervisor, such as page faults, that should not be reflected in target performance measurements. We create a virtual hardware TSC for the target that the hypervisor advances only when the target is running. We accomplish this with existing hardware virtualization support to adjust the target's virtualized TSC. Thus, within the simulation, the hardware reports target time.

3.3 Implementation

This section describes challenging and novel parts of 0sim's implementation. Note that 0sim only runs in the platform kernel; 0sim can run any unmodified target kernel. We implement 0sim as a modification to Linux kernel 4.4 and the KVM hypervisor. The kernel changes comprise about 4,100 new lines of code and 770 changed lines, and for KVM 400 new lines and 12 changed lines. By comparison, the gem5 simulator is almost 500,000 lines of code [31].

Memory Compression

We modify Linux's Zswap memory compression kernel module [4] to take advantage of data-obliviousness.

First, we modify Zswap to achieve compression ratios of 2^{15} for data in the common case: we represent zero pages with a single bit in the *zbit* bitmap, indicating whether it is a zero page or not. In practice, page tables and less-compressible pages (e.g., text sections or application metadata) limit compressibility, but ideally 1TB can be compressed to 32MB. In zbit, each page gets a bit. When selected for swapping by the platform, an all-zero target page will be identified by Zswap and compressed down to 1 bit in zbit. When the swapping subsystem queries Zswap to retrieve a page, it checks zbit. If the page's bit is set, we return a zero page; otherwise, Zswap proceeds as normal. Internally, zbit uses a radix tree to store sparse bitmaps efficiently.

Second, we observe that even non-zero pages can still be significantly compressed and densely stored. Zswap uses special memory allocators call "zpools" to store compressed pages. The default zpool, *zbud*, avoids computational overhead and implementation complexity at the expense of memory overhead. It limits the effective compression ratio to 2:1 and stores 24 bytes of metadata per compressed page [4].

We implement our own zpool, *ztier*, that significantly improves over zbud. All memory used by ztier goes toward storing compressed pages. It reduces metadata with negligible computational overhead by making use of unused fields of `struct page` and maintaining free-lists in the unallocated space of pages in the pool. Moreover, it supports multiple allocation sizes, leading to higher space efficiency. Thus, ztier achieves higher effective compression ratios than Linux's zbud at the expense of implementation complexity. For example, using zbud for a 500GB memcached workload on unmodified Zswap requires 294GB of memory. With zbit, zbud requires 15GB of RAM to store the compressed pages. In contrast, ztier consumes less than 6GB.

Overall, with our modifications a target page with predetermined content takes 66 bits of platform memory: 64 bits for a page table entry, 1 bitmap bit, and 1 bit amortized for upper levels of the page tables.

These optimizations allow the platform to keep most target state in memory, but a swap device is still needed since not all simulation state is compressible, and state may still need to overflow to the swap device. The amount of needed swap space depends heavily on the workload. Linux assigns swap space before attempting to insert into Zswap, even though it does not actually write to the swap space if Zswap is used. Thus, we thin-provision the swap space using device mapper [2] to look much larger than it actually is. We found that 1TB of swap space was sufficient for most of our workloads. Workloads with high churn or low compressibility required 2-3TB of swap space.

Shadow Page Tables

As mentioned in Section 3.2, 0sim uses shadow page tables to decouple the size of the target address space from the amount of physical address bits in the platform processor. The hypervisor reads the guest kernel's

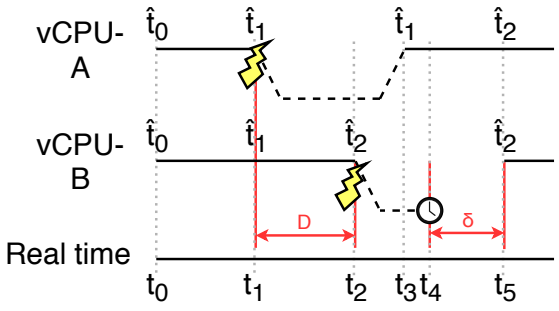


Figure 3.2: Example of proposed DTS mechanism. Both vCPUs start at target time $\hat{t}_0$ at platform (real) time t_0. At time t_1, vCPU-A pauses due to a trap or interrupt to the hypervisor, but vCPU-B continues to execute. At time t_3, vCPU-A continues, while vCPU-B is paused. At time t_4, vCPU-B is ready to run again but is ahead by D time units, so we delay it by δ time units to give vCPU-A time to reach target time $\hat{t}_2$. At platform time t_5, vCPU-A has caught up, so vCPU-B is allowed to run.

page tables and constructs the shadow page tables which are used by the hardware to translate guest virtual addresses to host physical addresses. Hardware never sees guest physical addresses. Thus, the target physical and virtual address spaces are as large as the platform virtual address space (48 bits, rather than 39 bits, on our machine).

When the platform has enough physical address bits, 0sim can optionally use hardware-based nested paging extensions [29]. Nested paging does not have the space overhead of shadow page tables, and is faster because the hypervisor is not on the critical path of address translation. However, the added simulation speed comes at the expense of some accuracy, as 0sim cannot account for the overhead of nested paging, which happens transparently in the hardware. Thus, there is a tradeoff between simulation speed and accuracy; for more accuracy, one can disable nested paging extensions.

Time Virtualization

0sim virtualizes the `rdtsc` x86 instruction, which returns a cycle-level hardware timestamp counter (TSC). Each physical core has an independent TSC, and the Linux kernel synchronizes them at boot time. Most Intel processors have the ability to virtualize the TSC so that the guest TSC is an offset from the platform TSC of the processor it runs on [77]. 0sim adjusts this offset per-vCPU to hide time spent in the hypervisor rather than executing the target.

Virtualization itself has associated overheads. For example, the hypervisor emulates privileged instructions and I/O operations from the target kernel. We modify KVM to hide most virtualization overhead from the simulation. We record platform TSC values whenever a vCPU stops (i.e., when target time pauses). Before the vCPU resumes, we read the platform TSC again and offset the target TSC by the elapsed time.

Preserving timing in multi-core simulations presents an additional challenge because the hypervisor executes simulated cores concurrently on different platform cores. Since vCPUs can be run or paused independently by the hypervisor, their target TSCs may become unsynchronized. This can be problematic if timing measurements may cross cores or be influenced by events on other cores (e.g., synchronization events, responses from server threads). For some use cases, this can cause measurement inaccuracy.

In this section, we propose a solution to this problem, which we call *Dynamic TSC Synchronization* (DTS). While we have implemented DTS in `0sim`, we leave it off for all the experiments in this paper because it increases simulation time and our current implementation can sometimes cause instability; more evaluation is needed before it should be used. While we have observed drift in our experiments, we find that `0sim` is still accurate enough for many use case, as shown in sections 3.4 and 3.6.

DTS works as follows: To prevent excessive drift, `0sim` delays vCPUs that run too far ahead to give other cores a chance to catch up. Specifically, let $\hat{t}_v$ be the target TSC of vCPU v. `0sim` delays a vCPU c by descheduling it from running for δ time units if it is at least D time units ahead of the most lagging target TSC:

$\hat{t}_{min} := \min_v \hat{t}_v$ ▷ Most lagging target TSC
if $\hat{t}_{min} < \hat{t}_c - D$ **then** delay c by δ
else run c

Periodic events such as timer interrupts prevent cores from running ahead indefinitely. Figure 3.2 walks through an example. Generally, simulator speed will decrease as D decreases and as the number of simulated cores increases. Users can adjust the parameters D and δ as needed.

DTS has many desirable properties. Most importantly, it bounds the amount of drift between target TSCs to the threshold D. This means that it is possible to get more accurate measurements by measuring longer, since error does not accumulate. Moreover, it does not cause target time to jump or go backwards.

LAPIC Timer Interrupts. Operating systems commonly use the arrival of timer interrupts as a form of clock tick. For example, the Linux kernel scheduler and certain synchronization events (e.g., `rcu_sched`) perceive the passage of time using `jiffies`, which are measured by the delivery of interrupts. We virtualize the delivery of timer interrupts by delaying their delivery to the target kernel until the appropriate guest time is reached on the vCPU receiving the interrupt. We empirically verified that the target perceives an interrupt rate comparable to the platform.

Limitations. Because many events (e.g., I/O) are emulated by the hypervisor, there is no clear way to properly know how much time they would take on native hardware. Rather than guess or use an arbitrary

Table 3.1: Specifications of test platforms for 0sim.

Machine	OS	CPU	DRAM	Boot Disk	Swap Disk
wk-old	CentOS 7.6.1810 Linux kernel 4.4.0	Intel Core i7-4770K, 3.50GHz Haswell 2013 39-bit physical address 48-bit virtual address	31GB DDR3 1600MHz	1TB HDD SATA 6GBps 7200RPM	2TB HDD SATA 6GBps 7200RPM
wk-new	CentOS 7.6.1810 Linux kernel 4.4.0	Intel Core i7-6700K, 4.00GHz Skylake 2015 39-bit physical address 48-bit virtual address	62GB DDR4 2133MHz	100GB SSD shared with swap SATA 6GBps 555MBps seq read 500MBps seq write	10TB thin-provisioned device backed by 365GB SSD partition
server	CentOS 7.6.1810 Linux kernel 4.4.0	2x Intel E5-2660v3, 3.00 GHz Haswell 2014 46-bit physical address 48-bit virtual address	160GB DDR4 2133MHz	1.2TB HDD SAS 6GBps 10000RPM	10TB thin-provisioned device backed by 480GB SSD
baseline (AWS)	RHEL 8.0.0 Linux kernel 5.1	Intel E7-8880v3, 2.3 GHz Haswell 2015 46-bit physical address 48-bit virtual address	3904GB Frequency Unknown	30GB SSD AWS EBS gp2	N/A

constant, we opt to make such events take no target time; effectively, the target time is paused while they are handled by the hypervisor. As a result, 0sim is not suitable for measuring the latency of I/O events, though it can measure the CPU latency of events in I/O-bound processes. This is similar to other simulators [31].

Our scheme does not account for microarchitectural and architectural behavior changes from virtualization. Context switching to the platform may have microarchitectural effects that may affect target performance, such as polluting the caches or the TLB of the platform processor. For example, after the hypervisor swaps in a page from Zswap, the target may take a TLB miss, which is not accounted for.

In addition, 0sim does not perfectly preserve hardware events. In particular, Linux uses multiple sources of time, including processor timestamps (e.g., `rdtsc`), other hardware clocks (e.g., to track time when the processor sleeps), and the rate of interrupt delivery. 0sim only virtualizes processor timestamps and LAPIC timer interrupts.

In practice, we find that 0sim preserves behavior well, as we show in Section 3.4. However, targets see a higher-than-normal rate of I/O interrupts (e.g., networking or storage), which can lead to poor performance and crashes of I/O intensive workloads.

3.4 Simulator Validation

Before showing case studies in Section 3.6, we demonstrate that 0sim is accurate enough to be useful for reproducing scalability issues, prototyping solutions, and exploring system behavior. Since 0sim does not modify the data structures or algorithms of the target, it automatically preserves non-temporal metrics, such as the amount of memory consumed.

Methodology

0sim runs on a wide range of commodity hardware, from older workstations to servers. The specifications of our test platforms can be found in Table 3.1. `wk-old` is a 6-year-old workstation machine with 31GB of DRAM. `wk-new` is a 4-year-old workstation machine with 64GB of DRAM. Both machines cost around $1000-2000 when originally bought. `server` is a server-class machine with 160GB of DRAM. These machines all cost orders-of-magnitude less than a huge memory machine or prolonged rental of cloud instances.

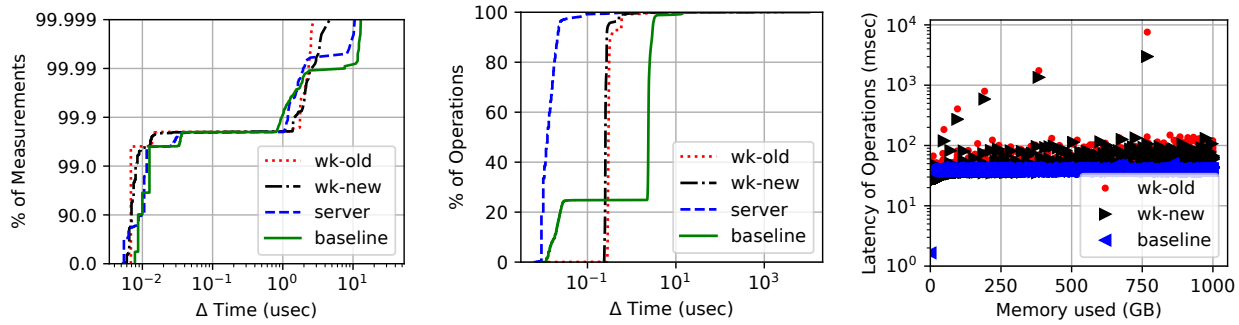
We set the CPU scaling governor to `performance`. Hyperthreads and Intel Turbo Boost are enabled everywhere for consistency because the server testbed we used does not have a way to disable them.

In all simulations, the target OS is CentOS 7.6.1810 with Linux kernel v5.1.4 since the stock CentOS kernel is several years old. We disable Meltdown and Spectre [86, 93] mitigations because they cause severe performance degradation when the host is overcommitted.

To collect metrics from the simulation, we export an NFS server from the platform to the target. This has reasonable performance and does not introduce new performance artifacts. We provide workloads, such as memcached and redis, with all-zero data sets. We modify microbenchmarks, like memhog, to use all-zero values. In all experiments with server applications (e.g., memcached), we run the client program driving the workload in the same VM as the server. Otherwise, I/O virtualization quickly becomes the bottleneck, so measurements are actually measuring aspects of the hypervisor, not the target.

All multi-core simulations have 8 simulated cores on desktop-class machines and 6 simulated cores on server-class machines. We found that even unmodified KVM is unable to boot multi-terabyte virtual machines with more than 6 cores on `server` when the platform is overcommitted. We believe this is because KVM's emulated hardware devices use real (platform) time, and the overhead of running very huge machines causes hardware protocol timeouts. In the future, we would like to extend 0sim's virtualization of time to address this. In the meantime, we expect 6 simulated cores to be enough to exercise multi-core effects of software.

Our comparison baseline is direct execution (not simulation) on an AWS `x1e.32xlarge` instance with 3904GB which costs $26.818 per hour [13]. This is the largest on-demand cloud instance we could find. We have run simulations up to 8TB, but here we use a maximum size of 4TB for comparison with the baseline. While 0sim simulates the performance of a *native* execution (excluding time spent in the hypervisor), the AWS instance is a virtual machine, so the overheads of virtualization are present, though it is not overcommitted.



(a) (b) (c)

Figure 3.3: (a) CDF of Δ Time between subsequent calls to `rdtsc` in simulations on different host machines. Note the log scale. (b) CDF of latency to touch and fault pages in simulations on different host machines. (c) Latency of sets of 100 insertions to memcached in simulations on different host machines.

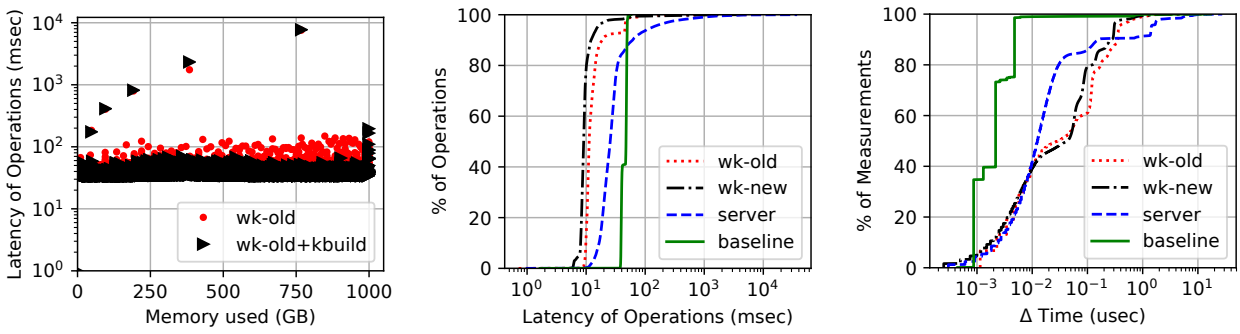
Single-core Accuracy

We first evaluate the accuracy of 0sim for single-core targets. To measure how well 0sim hides hypervisor activity from the target, we record the differences between subsequent executions of `rdtsc`. Figure 3.3a shows the CDF of timestamp differences. For all targets and the baseline, there are three main patterns: first, almost all measurements are less than 10ns, which is as fine-grained as `rdtsc` can measure and corresponds roughly to the pipeline depth of the processor. Second, there is a set of measurements of about 3µs. These correspond to the page faults from storing the results of the experiment. Finally, some measurements fit neither pattern and correspond to other system phenomena such as target interrupt handlers and times when the target kernel scheduler takes the processor away from the workload. 0sim hides hypervisor activity to closely approximate the baseline behavior on all three simulation platforms.

Conclusion 1: 0sim is able to hide idle hypervisor activity, such as servicing interrupts, from the target at the granularity of tens of nanoseconds up to the 99%-tile of measurements.

To validate that 0sim preserves the accuracy of the target OS, we run a workload that `mmap`s and sequentially touches all target memory. This causes significant kernel activity, as the kernel must handle page faults and allocate and zero memory. Likewise, significant hypervisor activity must be hidden from the target. Figure 3.3b shows the resulting time per page. Note that the `server` platform has much larger caches, aiding its performance. We see that 0sim is able to roughly preserve the time to touch memory, though we note that 0sim is not intended for such fine-grained measurements. The latency for the baseline machine is much higher (about 2.5µs) for 75% of operations because it includes hypervisor activity, such as allocating and zeroing pages, across NUMA nodes. In the simulations, hypervisor activity causes new pages to be cached, hiding inter-NUMA-node latency from the target.

Conclusion 2: In the presence of significant platform activity, 0sim is able to preserve timing of events to within 2.5µs,



(a) (b) (c)

Figure 3.4: (a) Latency per 100 sequential insertions to memcached while host is idle and doing a kernel build. (b) CDF of latency per 100 sequential insertions to memcached in multi-core simulations on different host machines. (c) CDF of Δ Time between subsequent memory accesses in simulations on different host machines for a workload with poor memory locality.

though it does not model NUMA effects.

To validate that 0sim can simulate more realistic workloads accurately, we simulate a workload that fills a large memcached server. The keys are unique integers, while the values are 512KB of zeros. We measure the latency of sets of 100 insertions in a 1TB workload. Memcached is implemented as a hashmap, so the time to insert is roughly constant for the entire workload. Figure 3.3c shows that 0sim preserves both the constant insertion time of memcached and the latency of requests compared to the baseline. The linear (note the logarithmic scale) trend of points at the top of the figure correspond to memcached resizing its hashmap, which blocks clients because they time-share the single core. This trend is not present in the baseline which is a multi-core machine and does resizing concurrently on a different core.

Conclusion 3: For real applications on single-core targets, 0sim is able to accurately preserve important trends, in addition to preserving high-level timing behavior.

Conclusion 4: 0sim produces comparable results when run on different platforms, down to scale of tens of microseconds.

To evaluate the sensitivity of simulation results to activity on the platform, we rerun the previous memcached experiment with a Linux kernel build running on the platform. Figure 3.4a shows the results on `wk-old` compared with the corresponding measurements from above. Despite the significant activity on the platform, the target sees similar results and trends to those collected above.

Conclusion 5: 0sim masks platform activity well enough to hide significant activity from the target.

Multi-core Accuracy

We evaluate 0sim's accuracy when running multi-core targets. A memcached client is pinned to one core and measures the latency of requests to a multi-threaded server not pinned to any core. Figure 3.4b shows a CDF

Table 3.2: Observed aggregate compressibility ratio for various workloads. NAS CG (class E) runs with its natural dataset, a sparse matrix. Metis runs a matrix multiplication workload.

Workload	Platform	Target	Compressibility
memcached	62GB	1TB	215:1
redis	160GB	1TB	231:1
Metis	160GB	4TB	327:1
NAS CG	31GB	500GB	16:1

of the measured latencies. The results are noisier than for the single-core simulations, as expected. 0sim is able to preserve the constant-time behavior of memcached, even without dynamic TSC synchronization. We believe the tail events for `wk-old` and `wk-new` represent increased jitter from multi-core interactions such as locking. We believe the long tail for `server` is due to the use of Intel nested paging extensions (EPT): nested page faults are hidden from the hypervisor, so we cannot adjust for them, as section 3.3 notes. Note that we measure 100 requests together, accumulating all of their cache misses, TLB misses, guest and host page faults, and nested page walks. One can obtain more accurate measurements by disabling EPT.

Conclusion 6: While 0sim produces less accurate results for multi-core targets, important trends and timing are preserved.

Worst-case Inaccuracy

To measure the worst-case impact of microarchitectural events, we run a workload with poor temporal and spatial locality that touches random addresses in a 4GB memory range. It incurs cache and TLB misses and both platform and target page faults, which are expensive since the host is oversubscribed. Figure 3.4c shows that these artifacts result in significant latency visible to the simulation. Around 90% of these events have a latency of 500ns or less, corresponding to the latency of cache and TLB misses in modern processors. Also, the server machine shows fewer such events, corresponding to its larger caches and TLB.

Conclusion 7: Experiments that measure largely microarchitectural performance differences such as TLB and cache misses may be inaccurate on 0sim.

3.5 Data-obliviousness and Speed

To achieve reasonable simulation performance, 0sim requires that the target have good compressibility. We believe this includes a large class of useful workloads. Table 3.2 reports the aggregate compressibility ratio for a few example workloads from our experiments, which is the average compressibility of all pages the kernel attempted to insert into Zswap (this is not the same as the ratio of platform memory to target memory). Note that an aggregate compressibility ratio of only 20:1 represents a 95% saving in memory usage over the

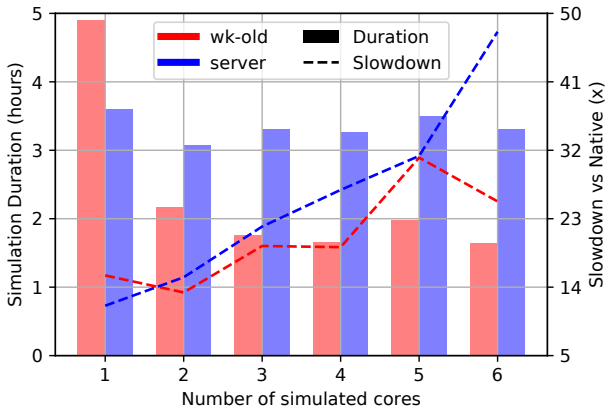


Figure 3.5: Simulation duration and slowdown compared to native execution (computed from TSC offset) for 1TB memcached workload as number of simulated cores varies.

course of the workload. A few results deserve comment. Metis is an in-memory map-reduce framework [82]; unfortunately, it crashes mid-way through huge workloads, highlighting a need to test systems software on huge-memory systems. Also, NAS CG runs unmodified with its standard data set (a sparse matrix), which is compression-friendly but not data-oblivious. Overall, these results show that 0sim is able to vastly decrease the memory footprint of huge data-oblivious workloads, making huge simulations feasible.

Data-obliviousness is critical to simulation performance and has some role in accuracy. We run an experiment in which we turn off Zswap, relying entirely on swapping. A 1TB memcached workload takes 3x longer to boot and 10x longer to run. Worse, the overhead is so high that despite 0sim's TSC offsetting scheme, overhead still leaks into the simulation and leads to target performance degradation proportional to the size of the workload.

Simulation speed depends heavily on the workload, platform, and number of simulated cores. Figure 3.5 shows the simulation speed of a 1TB memcached workload as the number of simulated cores increases. Generally, overhead increases with the number of simulated cores, but runtime may decrease due to improved workload performance.

In our experiments, we generally observe between 8x and 100x slowdown compared to native multi-core execution. For reference, this is comparable to running the workload on a late-1990s or early 2000s processor [125]. Architecture simulations often incur slowdowns of 10,000x or worse [31]. We found that workloads with heavy I/O are slower due to I/O virtualization.

Simulator performance degrades gracefully as platform memory becomes increasingly overcommitted. Users can balance simulation speed and scalability testing by varying the amount of target physical memory. In practice, we find that simulation size is limited by hard-coded limits and software bugs, rather than memory capacity. For example, KVM-QEMU does not accept parameter strings for anything larger than

7999GB. With engineering effort, such limitations can be overcome. Overall, we find that 0sim makes huge multi-core simulations of data-oblivious workloads feasible and performant.

3.6 Case Studies

0sim is useful both for prototyping and testing and for research and exploration. We give case studies showing how we debugged scalability bugs in memcached, explored design space issues in Linux memory fragmentation management and page reclamation, evaluated a proposed kernel patchset, and reproduced known performance issues.

Development

The ability to interact with 0sim workloads proved invaluable. While running experiments, memcached returned an unexpected out-of-memory error after inserting only two 2TB of data out of 8TB. To understand why memcached was misbehaving, we started an interactive (albeit slow) debugging session on the running memcached instance. We found that memcached's allocation pattern triggered a pathological case in `glibc`'s `malloc` implementation that led to a huge number of calls to `mmap`. This caused allocations to fail due to a system parameter that limits per-process memory regions. Increasing the limit resolves the issue.

This incident demonstrates that 0sim is useful for finding, reproducing, debugging, and verifying solutions for bugs that only occur on huge-memory systems. A lead developer of memcached was unsure of the problem because they had not tried memcached on a system larger than 1.5TB. We hope that 0sim can better prepare the systems community for the wider availability of huge-memory systems.

Exploration

We give two case studies in the Linux kernel demonstrating how 0sim can be useful for design space exploration.

Memory Fragmentation

Memory fragmentation at the application level and at the system level has been extensively studied in prior literature [9,22,25,27,52,64,68,73,80,91,108,114,115,115–117,121,128]. However, we are aware of little work addressing the effects of fragmentation in huge-memory workloads. Some have suggested that huge-memory workloads do not suffer extensively from fragmentation [21]. 0sim is well-suited for studying such workloads and their system-level effects.

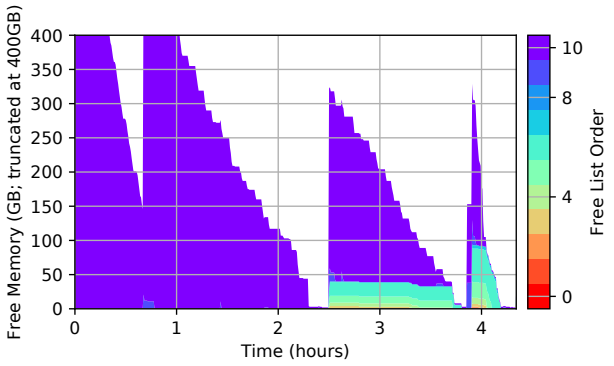


Figure 3.6: Amount of memory in each kernel page allocator free list in mix workload. The y-axis is truncated to 400GB to show more detail. More red indicates high fragmentation.

The Linux kernel allocator is a variant of the buddy allocator [68, 85, 117] and uses different free lists for different sizes of contiguous free memory regions. Specifically, there is a free list for each *order* of allocation, where the order-n free list contains contiguous regions of 2^n pages. The kernel merges free regions to form the largest possible free memory regions before adding them to the appropriate free list. Thus, if a large percentage of free pages is in the low-order free lists (closer to 0), memory is highly fragmented.

Methodology. We record the distribution of pages across free lists over time in the Linux v5.1.4 physical memory allocator. We simulate a 1TB Redis instance in isolation with snapshotting enabled. Redis periodically snapshots its contents: the Redis process forks, and the child writes its contents to disk, relying on kernel copy-on-write, and terminates, while the parent continues processing requests.

We then simulate a mixed workload: a redis client and server pair are used to represent a typical key-value server found in many distributed applications; a Metis workload represents a concurrent CPU and memory intensive computation; and a `memhog` workload modified to pin memory and be data-oblivious mimics high-performance I/O drivers that use large pinned physical memory regions for buffers [50]. These applications each receive 1/3 of system memory.

Results. Running alone, Redis does not suffer from fragmentation, but in the presence of other workloads it does. Figure 3.6 shows the amount of free memory in each buddy list throughout the mix workload. More purple (top) indicates more large-contiguous free physical memory regions, whereas more red (bottom) indicates that physical memory is highly fragmented. Each time free memory runs low, fragmentation degrades for subsequent portions of the workload: before time 2.5h, there is little fragmentation, but after 2.5h, almost 40GB of free memory is in orders 8 or lower, and after 4.2h, almost 100GB is in orders 8 or lower. Note that order 9 or higher is required to allocate a huge page. Upon closer inspection, we see that while most

regions are contiguous, many individual base pages are scattered around physical memory. These pages represent some sort of "latent" fragmentation that persists despite the freeing and coalescing of hundreds of gigabytes of memory. This suggests that any true anti-fragmentation solution must also deal with this "latent" fragmentation.

The above results deal with *external fragmentation* – that is, fragmentation that causes the waste of unallocated space. While running our experiments, we also discovered that for some workloads, *internal fragmentation* (wasted space *within* an allocation) is a problem at the application-level. Specifically, we observed that in some cases, a 4TB memcached instance could only hold 2-3TB of data due to pathological internal fragmentation. If the size of values inserted does not match memcached's internal unit of memory allocation, memory is wasted. This wastage increases proportionally to the size of the workload, so it becomes problematic for multi-terabyte memcached instances. We had to carefully tune the parameters of memcached to get acceptable memory usage. These observations also suggest that internal fragmentation may be a more important problem on huge-memory systems.

Page Reclamation.

Datacenter workloads may overcommit servers [46] to improve efficiency. A page reclamation algorithm satisfies kernel allocations when memory is scarce. In Linux, *direct reclamation* (DR) satisfies an outstanding allocation that is blocking userspace (e.g., from a page fault), while *idle reclamation* (IR) happens in the background when the amount of free memory goes below a threshold. Reclamation on huge-memory systems can be computationally expensive because it scans through billions of pages to check for idleness, potentially offsetting any gains made from the additional available memory. Google and Facebook both use in-house IR solutions to achieve better memory utilization efficiently [46]. Using `0sim`, we measure the amount of time each algorithm spends per reclaimed page and explore policy modifications to the DR algorithm to attempt to make it more efficient.

Methodology. We run a workload that hogs all memory and sleeps. Then, we run a memcached workload that only performs insertions into the key-value store. This causes idle and direct reclamation from the hog workload. We instrument Linux to measure the time spent in idle and direct reclamation and the number of pages scanned and reclaimed.

Results. The top of Table 3.3 ("Idle" and "Direct") shows that DR costs 7µs per reclaimed page, whereas IR costs 3.5µs per reclaimed page but runs about 5 times longer. This makes sense; DR blocks userspace

Table 3.3: Time spent and pages scanned and reclaimed with different reclamation policies and modes.

Mode	CPU Time (s)	Scanned	Reclaimed
Idle	24	25,637,891	6,631,878
Direct	5	2,473,659	706,596
Idle 4x	21	19,594,007	5,657,472
Direct 4x	7	6,382,659	1,695,243

Table 3.4: Number of chunks and amount of time spent initializing and freeing memory during boot with ktask.

machine	cores	memory	chunks	init time	free time
wk-new	1	1TB	8035	3.5s	2.6s
wk-new	8	1TB	8035	1.1s	1.0s
server	1	4TB	32224	16.2s	13.1s
server	20	4TB	32224	2.5s	3.4s
no-ktask	20	4TB	1	18.2s	15.2s

execution, so it is optimized for latency, rather than efficiency. Thus, it stops as soon as it can satisfy the required allocation, whereas IR continues until a watermark is met.

We hypothesized that DR would run less frequenctly if it were more efficient. We modify DR to reclaim four times more memory than requested. Table 3.3 (Idle 4x and Direct 4x) shows that direct and IR now both spend about 4µs per reclaimed page. Direct reclaim consumes about 2s more than before but runs about 36% less often, decreasing overall reclaimation time by 1s. This suggests that on continually busy systems, applications that can tolerate slightly longer latency may benefit from our modification.

Reproducing and Prototyping

0sim can be used to reproduce known scalability issues and prototype fixes for them. We demonstrate this with two case studies in the Linux kernel.

ktask scalability.

The proposed *ktask* patchset parallelizes CPU-intensive kernel-space work [44, 81], such as `struct page` initalization. Using 0sim, we reproduce and extend developer results for the ktask patchset.

Methodology. We apply the patchset [81] to Linux kernel 5.1 and instrument the kernel to measure the amount of time elapsed during initialization using `rdtsc`. During boot, the structs are first initialized; then, they are freed to the kernel memory allocator, making them available to the system. We also record the number of 32,768-page "chunks" used by ktask, which can be initialized in parallel with each other.

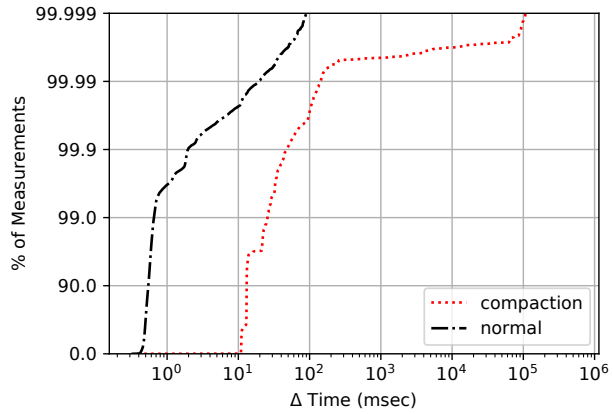


Figure 3.7: Latency of memcached requests in the presence and absence of continuous compaction. Note the log scale.

Results. Table 3.4 shows 3x and 5x improvement in initialization for 1TB machine with 8 cores and a 4TB machine with 20 cores, respectively. This is proportional to the results posted by the patchset author for a real 512GB machine [81]. However, even with ktask, page initialization is still expensive! On a 4TB machine, almost 6 seconds of boot time are consumed, whereas, for example, an availability of 5 "nines" corresponds to about 5 minutes of downtime annually. Some prior discussions among kernel developers [40, 42] have investigated eliminating `struct page` for some use cases, and our results suggest that `struct page` usage in the kernel is unscalable. Memory management algorithms are needed that do not scale linearly with the amount of physical memory.

Memory Compaction.

Using `0sim`, we reproduce and quantify memory compaction overheads. Huge pages and many high-performance I/O devices (e.g., network cards) [43, 45] require large contiguous physical memory allocations. The Linux kernel creates contiguous regions with an expensive memory compaction algorithm. Sudden compaction can produce unpredictable performance dips in applications, leading many databases and storage systems to recommend disabling features like Transparent Huge Pages, including Oracle Database [111], Redis [5], Couchbase [49], and MongoDB [102].

Methodology. We measure the latency of memcached requests in the presence and absence of compaction. Compaction usually happens in short bursts, making it hard to reproduce, so we modify the kernel to retry compaction continuously. Measurements with this modification approximate the worst-case during compaction on a normal kernel.

Results. Figure 3.7 shows the latency of memcached requests in the presence and absence of compaction for a 1 TB workload. Median latency degrades by 22x, while 99.999%-tile latency degrades by 10,000x, leading to occasional very-long latency spikes. Some of the tail effects are due to 0sim overhead, but Figure 3.3a shows that this overhead cannot cause such a large effect. This suggests that in a production system, compaction can lead to events that define service tail latency. 0sim can be used to further explore memory allocation and compaction policies for huge systems.

3.7 Future Work

In Section 3.1, we introduce and describe the Capacity Scaling Problem. More work is needed to understand the overheads and importance of various potentially unscalable algorithms. For example, many memory managers use "scanning" algorithms that iterate over all pages, e.g., memory reclamation, compaction, migration, deduplication, idleness checking. Recent prior work suggests that each of these algorithms may play an important role in future systems, as discussed in Section 2. Some key questions to answer include: (1) can these mechanisms work without any scanning or with an asymptotically lower amount of scanning? (2) are there new hardware features (e.g., improved performance counters, more efficient paging structures) that can enable more efficient memory management? (3) to what extent should MM on large machines look different from small machines (e.g., different policies vs entirely new design)?

Another avenue of future work is to improve 0sim itself by making it aware of more hardware features. Currently, 0sim enables large-scale simulations, but it assumes that memory homogeneous – it is oblivious to the underlying topology. For example, NUMA nodes and hetergeneous memory types are not exposed by 0sim to the target machine. Also, support for simulating I/O with low overhead could make many new workloads usable with 0sim. Finally, improving and validating the DTS mechanism can improve multicore simulation accuracy.

3.8 Conclusion

System scalability with respect to memory capacity is critical but under-studied. The memory usage, computational inefficiency, and policy choices of current systems are often unsuitable for huge systems. 0sim is a simulation platform designed to address this problem. It takes advantage of the data-obliviousness of many workloads to make their memory contents highly-compressible. 0sim runs on hardware that is easily available to researchers and developers, enabling both prototyping and exploration of system software. It accurately preserves behavior and trends. 0sim allowed us to debug unexpected behavior. By open-sourcing 0sim, we

hope to enable both researchers and developers to prepare system software for a world with terabyte-scale memories.

4 CBMM: FINANCIAL ADVICE FOR KERNEL MEMORY MANAGERS

I am the very model of a modern mem'ry manager.
I've information timely and of the highest caliber.
I know the kinds of mappings, and I guide the page table walker
from virtual to physical, in out-of-order processors.

I'm very well acquainted with matters economical.
I know the operations, both the fruitful and suboptimal.
About perf debugging I'm teeming with a lot of news
with many cheerful facts about the latency of huge page use.

— Adapted from The Pirates of Penzance, Gilbert and Sullivan, 1879

Acknowledgements. *This chapter contains published work from USENIX ATC 2022 co-authored with Bijan Tabatabai and Michael M. Swift* [*98*].

We thank the anonymous reviewers, Sujay Yadalam, and Yuvraj Patel for their time and insightful feedback on our paper. We thank the anonymous artifact reviewers and Anthony Rebello for their time spent testing our artifact. We thank Ashish Panwar for the help getting HawkEye set up. We thank Michael Marty who gave feedback on early versions of the project that became CBMM.

This work was funded by NSF grants CNS 1815656 and CNS 1900758.

Kernel MM mechanisms and policies comprise a complex set of tradeoffs that, when poorly navigated, lead to poor performance or unexpected behavior. For example, we found that for some workloads on Linux, a soft page fault lasting 25ms occurs every 100ms. This drastic tail latency is due to memory compaction or reclamation when attempting to allocate a huge page – a misnavigated tradeoff. Many applications would violate response latency objectives if one request per 100ms takes 25ms due to a page fault. As a result, Redis, MongoDB, and others advise users to disable Linux's Transparent Huge Page (THP) feature [5,49,102,111,143]. Table 2.1 lists other examples of MM policies and their potential pathologies.

On the other hand, datacenters prioritize *tail* latency as a key service-level metric, in addition to median latency and throughput [53]. System behavior must be consistent, i.e., low variance, without compromising performance metrics to satisfy service-level objectives and efficiency goals. Unfortunately, current MM designs often fall short of modern computing needs by exhibiting inconsistent, opaque behavior that is difficult to reproduce, decipher, or fix. These issues come from three key limitations.

First, kernel MM must predict workload behavior in an information-poor environment. Current MM designs rely on online measurements, particularly page table access/dirty bits and the frequency and location of page faults. Unfortunately, this information is expensive to collect and low bandwidth. For example, Google uses access bits to detect idle memory [35], but other work finds them insufficient to predict TLB miss overheads accurately [113], even though they can cost up to 11% of CPU cycles to collect [35]. Other data collection mechanisms induce additional page faults [36,62]. Recent work uses performance counters in kernelspace [113], but currently available counters are hardware-thread-oriented and do not provided the detailed spatial information useful for most MM policies.

Second, current MM designs often ignore the cost of various MM operations, leading to inappropriate policy decisions. For example, Linux allocates a huge page when a memory region is first touched; however, we find that allocating and zeroing a huge page costs 10^6 cycles in the best case. Thus, promoting a page that averts $\leqslant 10^6$ cycles worth of TLB misses and page faults actually *regresses* performance, but the kernel does not account for this cost.

Third, current MM designs are implemented as disjointly acting policies distributed throughout the kernel that are hard to debug. For example, code implementing Linux's huge page policies is scattered across more than eight files (and numerous functions), mixed with unrelated code. Users and developers observe erratic slowdowns without indication of what causes them or how to address them. They often resort to suboptimal coarse-grain solutions, such as disabling huge pages [5,49,102,111,143]. By distributing and obscuring policy-implementing code, current kernel MM implementations make it difficult for both kernel and userspace developers to decipher system behavior. This opaque system implementation and its consequent opaque behavior is a primary obstacle to improving kernel MM performance, consistency, and debuggability.

In search of a MM design that has consistent behavior, we designed Cost-Benefit MM (CBMM). CBMM reflects that all kernel MM operations have a cost and a benefit to userspace, and it estimates them using empirically based cost-benefit models to guide MM policy decisions. By explicitly modeling cost and benefit, CBMM is more cost-aware than current designs, so it makes fewer pathologically bad policy choices. Also, CBMM augments online statistics with offline-aggregated profiles to improve the quality of information available to the kernel. CBMM simplifies policy debugging and enables incremental performance improvement by centralizing models in a new kernel component: the *estimator*. To understand and fix anomalies, one must only understand the model inputs to determine the cause of a policy decision.

Our prototype implements models for huge page promotion, asynchronous page prezeroing, and eager paging [83], based on an in-depth analysis of huge page behavior and soft page faults. At runtime, they may make use of in-built empirically based assumptions (e.g., about average TLB miss latency), online

information (e.g., the current number of free pages), or offline-aggregated profile information (e.g., fine-grained information about huge page benefits). We focus on *first-party* datacenter workloads – software run by service providers in their own datacenters – as they are highly controlled and relatively stable over time, allowing better profiling and modeling [20,28,65,79,100,129,132,134].

CBMM improves system consistency; it nearly always has better tail latency than Linux or HawkEye, particularly on fragmented systems. It reduces the cost of the most expensive soft page faults by 2-3 orders of magnitude for most of our workloads, and reduces the frequency of 10-1000µs-long faults by around 2 orders of magnitude for multiple workloads. Meanwhile, it has competitive performance with Linux and HawkEye, a recent research system [113], for all the workloads we ran, and in the presence of fragmentation, CBMM is up to 35% faster than Linux on average – all while using no more huge pages than Linux or HawkEye in most cases.

4.1 Evaluating Current Behavior

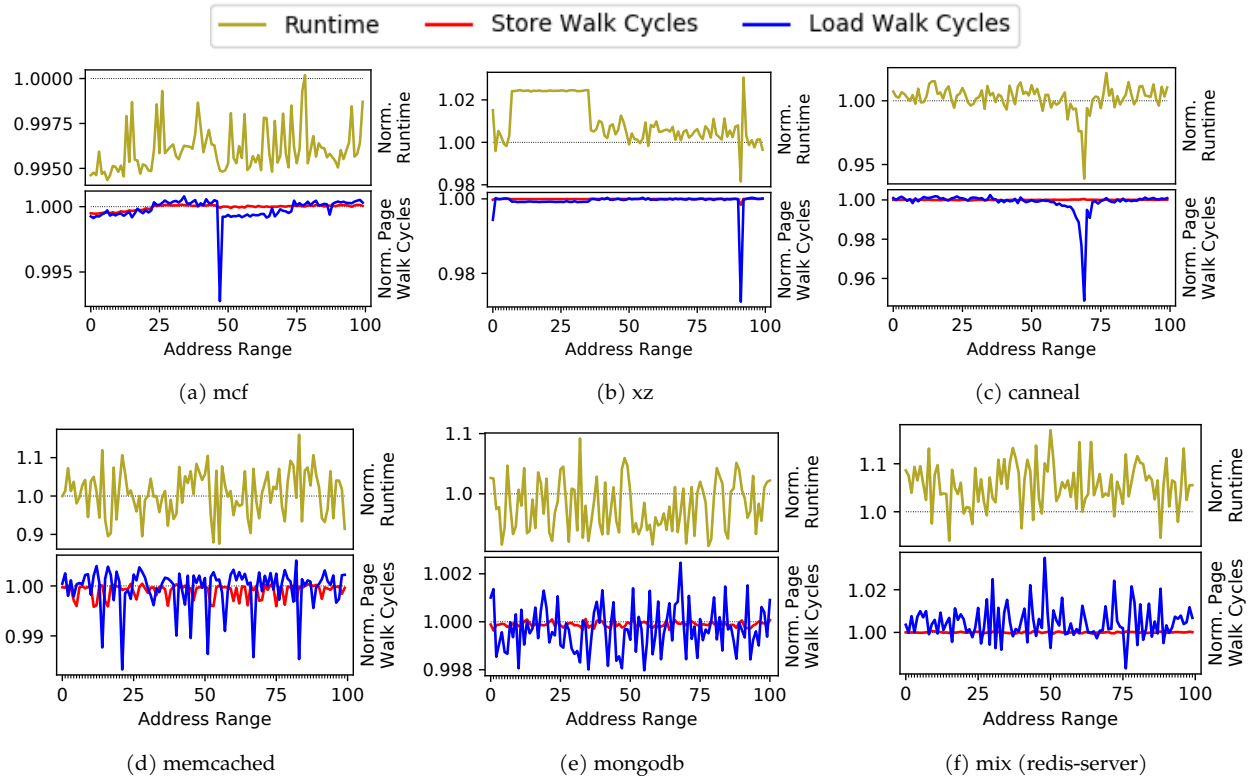
To quantify the extent of these challenges and inform our design, we do an in-depth analysis of two important kernel MM code paths, huge page management and page fault handling. Our experimental setup is described in Section 4.4.

Measuring Huge Page Benefits

Huge pages speed up many workloads, but nobody has quantified the impact of workload behavior on the amount of speedup it receives from huge pages. Thus, we measure the fine-grained benefit of huge pages as described in Section 4.3. To avoid invasive instrumentation and a detailed survey of workload implementations, we measure huge page benefits from the perspective of the kernel: for each workload, we divide the address space into 100 equally sized ranges, excluding unmapped regions, and repeatedly run the workload backing one range at a time with huge pages.

Figure 4.1 shows the results. Each point on the x-axis represents one range, such that the x-axis represents the virtual address space. The top y-axis shows the normalized performance compared to no huge pages. The bottom y-axis shows the normalized percentage of time spent in usermode page walks (i.e., TLB misses) for loads and stores.

The impact of huge pages varies extensively between workloads. `xz` and `canneal` primarily see improvements in *load* page walk cycles from backing particular regions of the address space corresponding to hot data structures. `memcached` and `mongodb` produce noisy results because of randomness in the workload. The magnitude of impact ranges from about 0.25% in `mongodb` to almost 7% in `canneal`.



(a) mcf (b) xz (c) canneal

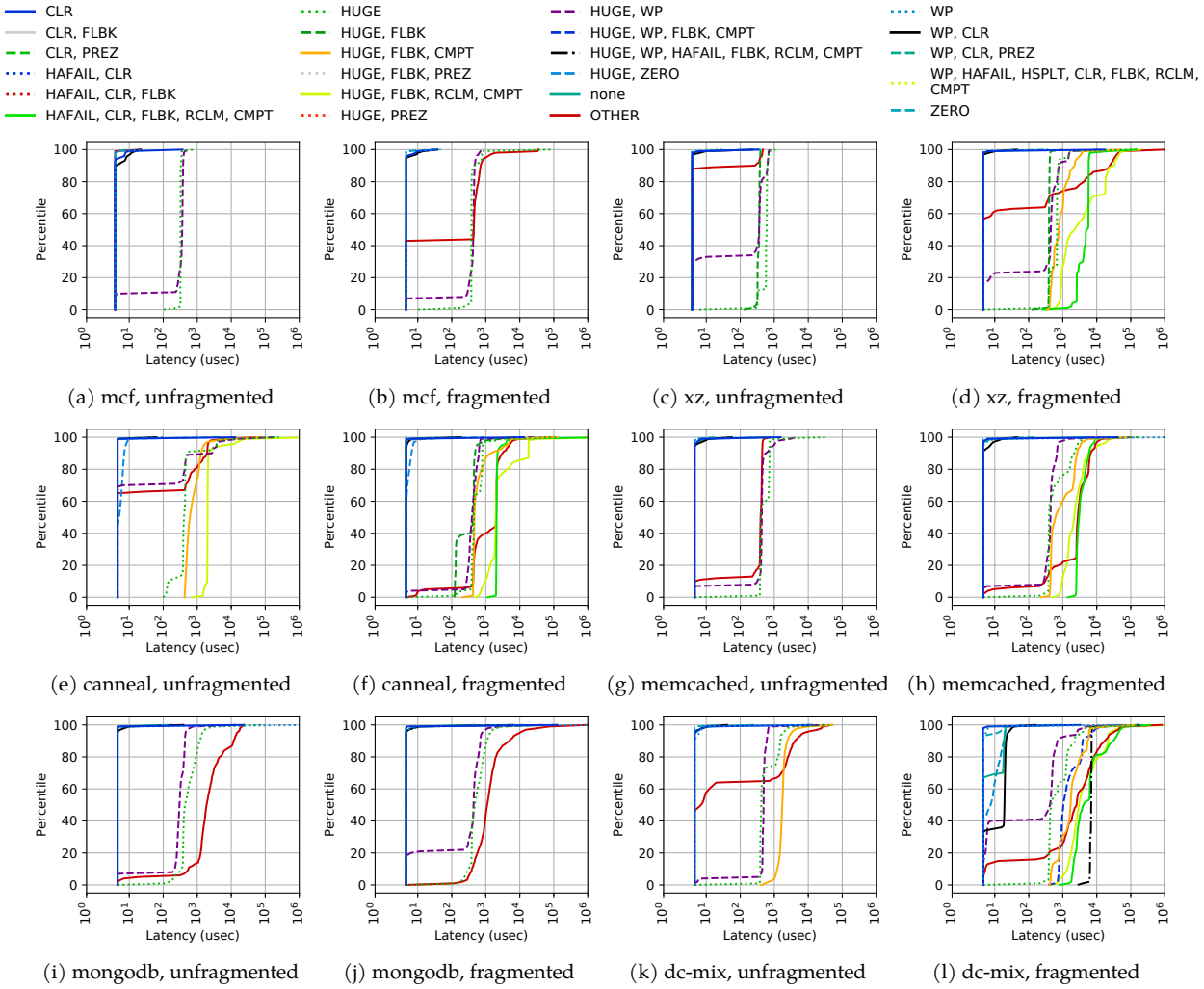
(d) memcached (e) mongodb (f) mix (redis-server)

Figure 4.1: Runtime and usermode cycles spent in page walks for each address range, normalized to no huge pages (lower is better). Note the varying y-axes.

Another benefit of huge page usage is fewer page faults. We found that they have only a minor contribution to performance (e.g., less than 1.2% of execution time for `canneal`).

Also, the relationship between runtime improvement and reduction in page walk cycles is not straightforward. For all workloads, runtime improvement is loosely correlated with either load or store page walk cycles. Strong effects on either load or store page walk cycles tended to be reflected in runtime, as seen in `xz` and `canneal`, but the magnitude of that effect varies. Small changes in page walk cycles often have no apparent effect on runtime.

Discussion Huge page impact varies greatly by workload, including the type and location of impacted memory accesses and the magnitude of impact. Additionally, the relationship between page walk cycles and runtime is complex, illustrating the challenge of huge page management given the limited, low-quality information available to the kernel at runtime such as CPU performance counters and page referenced bits. For example, `dc-mix` (not depicted) benefits from backing individual regions with huge pages, but when THP is turned on, it sees a net regression in performance due to the overhead of compaction. CBMM aims to mitigate this problem by supplying the kernel with higher-quality information.



(a) mcf, unfragmented (b) mcf, fragmented (c) xz, unfragmented (d) xz, fragmented

(e) canneal, unfragmented (f) canneal, fragmented (g) memcached, unfragmented (h) memcached, fragmented

(i) mongodb, unfragmented (j) mongodb, fragmented (k) dc-mix, unfragmented (l) dc-mix, fragmented

Flag	Description
`CLR`	Memory was zeroed (usually during allocation).
`CMPT`	Allocator used memory compaction.
`FLBK`	Allocator used a fallback path during the page fault.
`HAFAIL`	Attempted to allocate a huge page and failed.
`HSPLT`	A huge page was split into base pages.
`HUGE`	A huge page was mapped.
`PREZ`	Allocator allocated a prezeroed page (CBMM only).
`RCLM`	Allocator used direct reclamation.
`WP`	Page fault due to write to a write-protected page.
`ZERO`	A (shared) zero page was mapped.

(m) Subset of bitflags for page fault tracing.

Figure 4.2: CDF of Linux soft page fault latency by type of page fault. Not all page fault types occur in all workloads.

Soft Page Fault Latency Breakdown

We instrument Linux's page fault handler to trace sources of page fault latency. Page fault tracing allows us to characterize system-wide costs, such as the cost of zeroing memory. We identified a set of events that occur during page faults and associate each with a bitflag (Figure 4.2m). Our instrumentation records the

total time of the page fault, the time to allocate memory, and the time to clear/copy memory contents.

We record the flags and timing of all events longer than 10^4 cycles, and a count of shorter events, allowing us to compute the proportion of all page faults with each set of flags. We exclude hard page faults from our results, as they incorporate other kernel subsystems (e.g., block I/O, file systems). Our tracing records the total time to handle a page fault, but on x86 the handler can be interrupted in favor of another task, which inflates the latency of the page fault. This is rare in most workloads except `mongodb`, which uses a userspace asynchronous I/O framework and thread pool; even though a page fault handler may be descheduled for a while, userspace requests continue to make progress because of userspace threading.

Figure 4.2 shows the soft page fault latency breakdown for multiple workloads. For each distinct set of flags, the CDF of page faults with those flags is plotted. Note that the x-axis uses a log scale. The plot includes samples lower than the threshold by treating them as if they all took 10^4 cycles (in reality, most are faster than that). The figure shows results on a freshly booted, unfragmented system, which represents best-case performance; we also recorded results on fragmented system, and found them significantly worse for all workloads.

The results indicate three challenges current MM designs face. First, applications trigger a wide variety of kernel behaviors. Each of the 15 flag-sets of Figure 4.2 is a different combination of code paths. Second, different paths have very different latencies but are relatively consistent across workloads. For example, even in this best case, a huge page consistently takes hundreds of microseconds to be allocated (`HUGE` in the figure) due to zeroing overhead. Third, many pathological code paths execute that do not benefit applications. Most notably, a huge page allocation may invoke a fallback path (`FLBK`), which transitively invokes compaction (`CMPT`) or reclamation (`RCLM`). Worse, the fallback may fail (`HAFAIL`), resulting in a base page allocation after all. In `canneal` (Figure 4.2e) and `dc-mix` (Figure 4.2k), these fallback paths can take *dozens or hundreds of milliseconds*. In contrast, an Amazon search for "DRAM" completes in only 900ms from our office.

Discussion Linux's fallback algorithms are severely cost-unaware and make system behavior inconsistent: invoking compaction or reclamation almost certainly outweighs any benefits of using a huge page. Also, the high cost of zeroing suggests that memory prezeroing (Section 4.3) may be a useful optimization to make huge pages more useful. Currently, if an average TLB miss costs around 30 cycles, then a huge page must avert over 33,000 TLB misses to pay for itself. These results highlight the need for cost-aware MM policies.

4.2 Cost-Benefit Memory Management

We created the Cost-Benefit Memory Manager (CBMM), which has several goals:

- Improve kernel MM behavioral consistency,
- Match existing systems' performance,
- Improve the debuggability of policy decisions,
- Allow incremental improvement of individual policies.

Our key insight is that all MM decisions incur a cost against and provide a benefit to userspace. For example, huge page promotion averts TLB misses but may require zeroing or compacting memory. In CBMM, policy decisions follow the guiding principle that *userspace benefits must outweigh userspace costs*. By applying this principle uniformly, CBMM significantly improves consistency over Linux and HawkEye [113], while matching their performance. We design models for three important kernel MM policies: huge page promotion, asynchronous page prezeroing, and eager paging [83].

CBMM introduces a new component, the *estimator*, to the kernel. It estimates the cost and benefit of a given MM operation whenever a policy decision is needed. If $\texttt{cost} < \texttt{benefit}$, the kernel decides to execute the operation.

The estimator makes estimates based on empirically derived cost and benefit models. Models can optionally use live metrics and/or pre-aggregated profiling information. Such pre-aggregated information can mitigate the lack of high-quality online information. Meanwhile, CBMM explicitly estimates MM operation costs, improving cost-awareness.

In current MM implementations, policy decisions are scattered across the kernel, making it difficult to coordinate their actions and difficult to debug anomalous behavior. In contrast, CBMM invokes the estimator at decision points, which predicts the cost and benefit of taking an action. This centralizes decision making and explicitly marks policy decisions points. It also makes coordination between policies easier.

A key requirement of CBMM is that the system behavior can be modeled and/or profiled. This requirement holds for many first-party datacenter workloads, which often run with high redundancy for long amounts of time [20, 28, 65, 79, 100, 129, 132, 134], giving ample opportunity to observe and instrument a workload before applying policies to them.

The Estimator

In CBMM, the MM subsystem invokes the estimator at places in the code where policy decisions need to be made. We call these places in the code *decision points*. It uses models to estimate the cost and benefit of a particular MM operation and returns the estimates to the decision point, which executes the operation if *cost* $<$ *benefit*.

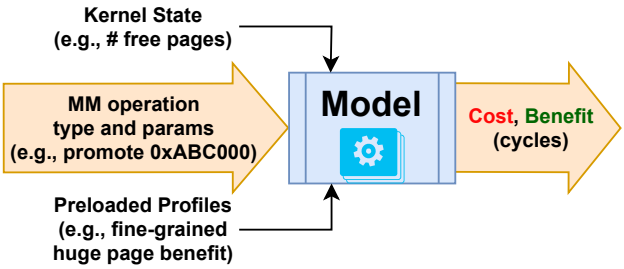


Figure 4.3: CBMM model inputs and outputs.

When a decision point invokes the estimator, it passes information to the estimator about the type and parameters of the operations. For example, the decision point would pass the address to consider promoting or a number of pages to attempt to prezero. The estimator acts as a black box that returns a cost and benefit estimate for the given MM operation and parameters. In CBMM, costs and benefits are computed in units of time saved or lost by userspace, which usually corresponds closely to user objectives. In particular, CBMM uses the *rate* of time saving/loss over some horizon, as many datacenter workloads run continuously.

Cost and Benefit Models

Internally, the estimator comprises a collection of cost models and benefit models for different MM operations. Each model is built out of simpler submodels that estimate one cost and/or benefit well; the submodel results are added to produce the overall result. This allows reuse of submodels for different decision points, simplifying implementation and leading to more consistent behavior across decision points. For example, our huge page cost-benefit models were useful in both the page fault handler and `khugepaged`, the background promotion daemon, and our model for estimating the cost of running a daemon could be used for multiple daemons in the future.

Concretely, models manifest as C code in the estimator (in the kernel); in Listings 4.4 - 4.6 (discussed further in Section 4.3), we show the models in our prototype of CBMM. Each (sub)model is a self-contained black box that takes information from the decision point, combines it with information from the ambient kernel state and *preloaded profiles* – files loaded into the kernel that supply information about application-specific behavior – and outputs an estimate, as shown in Figure 4.3. The additional input from the kernel state and preloaded profiles allows the models to be more context-aware and to make use of higher-quality information about workload behavior.

Performance Debugging Unlike current heuristics, CBMM isolates policies to specific cost and benefit models; their inputs and outputs can be observed, and they can be improved in a single place, easing

performance debugging in CBMM compared to Linux. A central idea behind CBMM's debuggability is the ability to observe and control the inputs to models. Thus, while models can make use of any kernel or hardware state, they should use only state that has an intuitive interpretation, rather than internal implementation metrics. For example, our huge page promotion model takes into account whether any prezeroed huge pages are available and uses a profile to determine the worth of promoting a page. In contrast, internal implementation metrics give limited information about the origin of their values and how to cause them to change, making bug fixing difficult; for example, Linux's page reclamation algorithm uses an obscure combination of page table bits, bit flags in the `struct page`, and what list a page happens to be on [18].

Model Development in CBMM can be done iteratively by beginning with a simple model and refining it as needed. For example, Listing 4.5 shows our asynchronous prezeroing model. Initially, we only accounted for the zeroing time of the daemon, but we found that this led to high lock contention on the allocator, so we refined the model to account for contention.

In designing our models, we found that benefits tend to be application-specific, whereas costs tend to be system-specific. For example, each application tends to benefit differently from huge pages, but the cost to allocate a huge page is application-independent and depends more on the state of the system allocator. As a result, our benefit models tend to use preloaded profiles, whereas our cost models tend to query kernel state.

Models necessarily make assumptions to simplify implementation and to make their execution cheaper than the actual MM operations. We based our assumptions on our empirical measurements, unlike many existing heuristics, which rely on intuitive simplicity or common-case optimization. For example, unlike Linux, CBMM does not blindly assume huge pages improve performance; rather, it incorporates the cost of promotion as measured by our experimental analysis and uses empirically derived profiles to estimate the benefit of promoting a particular memory region. Notably, CBMM improves system behavior even with imprecise profiles, as we will show in Section 4.4, making it practical to start with a simple model and refine it over time.

Preloaded Profiles

Different applications respond differently to MM policies, and kernels currently lack high-quality information with which to predict workload behavior. Preloaded profiles are files loaded into the kernel when starting a process (e.g., by a cluster manager) to provide models with information about a process's behavior. They allow the estimator to benefit from offline processing for particular policy decisions. In contrast, prior methodologies resort to measuring inaccurate and expensive proxy statistics such as page fault counts or page access bits.

In CBMM, preloaded profiles specifically provide *spatial, per-process information*; that is, they provide information about regions of a single address space at arbitrary granularity as small as a 4KB page. For example, a profile may specify per-region reduction in page walk cycles from use of huge pages, or a bit indicating whether a page is likely to be touched or not. Models can query this information when making cost and benefit estimates. For example, to estimate the benefit of using a huge page, a model may incorporate the number of averted page walk cycles, or to determine whether to eagerly allocate memory or use copy-on-write, a model may incorporate information about the likelihood of memory accesses. This structure for preloaded profiles, while simple, is quite useful because many MM policies make *spatial* decisions, such as whether/how to map/unmap/remap a memory region. However, CBMM's design is flexible enough to admit future extensions to profiles. For example, it may be desirable to account for phases of workload activity or to apply profiles at different granularities, such as per-thread or system-wide.

Profile Management. While CBMM still has benefits even when profiles are imprecise (see Section 4.4), changes to data structures or algorithms could result in changing memory reference patterns. Thus, a natural future extension of CBMM is automating profile generation and management. We give a brief description of automatic policy discovery in this section. In prior work, we explore in detail the potential role of cluster managers in policy discovery and deployment [97].

First-party datacenter workloads often run continuously and redundantly, so profiling could be automated and centralized at the cluster level. Thousands of identical tasks run across a WSC for hundreds of machine-years, and the workload mix changes incrementally as software teams update their services. Our target setting has relatively stable and homogeneous hardware and software. Most machines in the cluster are similar to a large number of other machines (not necessarily a majority) in both hardware and software mix. Many production WSCs satisfy these conditions [65,79,100,132,134]. Moreover, recent work from industry suggests a trend of large-scale profiling and centralized planning [88,95,129] and demonstrates the feasbility of such an approach.

We have done preliminary exploration and believe that the huge page methodology of Section 4.3 can be run in a distributed, automated manner by cluster managers. In particular, our methodology runs many identical copies of a workload with slight policy variations (e.g., regarding which parts of an address space should use huge pages). In a WSC, many identical copies of each workload are running already for availability or fault tolerance. A cluster manager can leverage this redundancy by instructing the kernels on different machines to apply slightly different policies. By collecting metrics from a large number of machines with slightly different policies, the cluster manager can estimate the impact of the policy differences and choose the policy that works best. Thus, the cluster manager can build a workload profile and keep it up-to-date in a

mostly automatic manner.

System Management

We implemented models directly in the kernel source, but in principle, they could be implemented via another mechanism, such as kernel modules. Our models are application-agnostic (but can be customized if needed, like existing code), so application developers do not need kernel access. Many service providers have kernel teams that could maintain this code. Meanwhile, profiles are application-specific, and application developers can use existing configuration/deployment systems to schedule/manage/store/secure/deploy profiles without special privileges. The kernel memory overhead of profiles depends on profile resolution/detailedness. In Section 4.4, our largest profile is $\sim$ 170KB/process and most profiles are $<$ 50KB/process.

Our implementation uses procfs files to load profiles, but any user-kernel communication mechanism could be used. Also, in principle one could load models through boot time configuration, similar to Facebook's SoftSKU system [129].

Discussion

CBMM addresses the (1) information-poverty, (2) cost-obliviousness, and (3) disjointed implementation of existing MM policy implementations, while other alternatives only partially address them. For example, interfaces such as `madvise` are coarse-grained, whereas workload memory access patterns can vary significantly within a region, as shown in Section 4.1. Merely disabling overly-aggressive policies, such as Linux's THP or defragmentation policies, harms workloads that require many huge pages, as we will see in Section 4.4. Additionally, it is difficult to modify existing policies to target different performance goals because their implementation is often distributed across many files, such as Linux's huge page policies. CBMM mitigates all three challenges by making costs and benefits explicit and centralizing policy decisions. Finally, more research is also needed to determine how far CBMM's design can be generalized to other areas of the kernel, such as scheduling, filesystems, or I/O management.

Similarities to Other Systems. VMware ESX Server explores MM techniques based on economic models by quantifying the value of idle memory and "taxing" processes for it [138]. Google and Meta both track and reclaim cold memory from processes, too [35, 88, 140]. Google's system centrally and empirically coordinates content migration to far-memory tiers (e.g., compressed memory) [88], while Meta's system relies on better metrics and acts locally on each machine. Google also profiles the lifetime of allocations to decrease memory fragmentation [95]. These approaches inspired our work; they use empirical measurements and MM-wide guiding principles to make MM decisions. Our work extends and generalizes this idea. Sriraman et al.

take a step in this direction by comprehensively profiling Meta's workloads and using the profiles to guide coarse-grained boot-time system tuning [129].

4.3 Implementation

We implement CBMM based on Linux 5.5.8 for three kernel MM policies: huge page management, asynchronous prezeroing, and eager paging [83]. We implement the estimator and its models, along with related debugging interfaces, code for parsing profiles, and other infrastructure in 1159 lines of C in the kernel in a new and self-contained file. Additionally, we add 87 lines of instrumentation throughout the page fault handler and page allocator for page fault tracing (see Section 4.1). We add 10 calls to the estimator throughout the MM subsystem; each is self-contained and consists of about 10 lines of code to initialize a struct, make a function call, and respond to estimates. Asynchronous prezeroing is implemented in a kernel module from HawkEye. We modify the module to run in a kernel thread and to query the estimator before running. Our version of the module is 196 lines of C. Our implementation is available at `https://github.com/multifacet/cbmm-artifact`.

Huge Page Management

Background Huge page support in current kernels can be either manual and automatic. A primary challenge is choosing memory regions to promote: the kernel must determine which memory regions would see enough benefit from huge pages.

Manual management allows applications to directly request huge pages for certain memory regions, but it requires modifying the applications, which is often impractical (e.g., Java does not expose a way to easily control memory allocation). Moreover, modern datacenter workloads are multi-programmed and diverse in behavior, requiring centralized coordination during resource allocation [87]. In contrast, automatic huge page management is a kernel feature that promotes application memory transparently to applications. This allows unmodified applications to benefit from huge pages but cannot make use of application-specific domain knowledge.

CBMM combines both the generality of automatic management and the application-specific knowledge of manual management. In contrast, current kernels either have only a manual system (e.g., Windows) or use simplistic heuristics to power an automatic system. For example, Linux's THP aggressively tries to promote on the first page fault to the huge page, potentially leading to memory bloat and increased page fault latency. FreeBSD waits for a specific percentage of the huge page to be touched before promoting. Various research

systems use a mix of page access bits, performance counters, LRU lists, and trial-and-error [87,113,142] with mixed success.

Model Listing 4.4 shows CBMM's model for huge page promotion. It is used in both the page fault handler and khugepaged to decided whether to promote a page. We built this model based on our analysis of huge page promotion overheads. It makes a number of simplifying assumptions when estimating both the cost and benefit, most notably that the cost is dominated by the allocation and zeroing time and that compaction and reclamation have a large fixed cost. We choose to ignore other costs in our model, such as caching, mapping changes, or potential memory bloat, but CBMM allows models to be iteratively improved over time.

```
void hpage_promo_model(u64 addr, mm_cost_delta *cost)
{
    // COST. Simplify using assumptions.
    // - Alloc is free if have free zeroed pages.
    // - Alloc cost is zeroing if have free unzeroed.
    // - Alloc cost is 2^32 if need to free mem.
    // - We don't care what node it is on.
    // - Constant prep costs (zeroing or copying), ~100us

    const u64 EXPENSIVE = 1ul << 32; // cycles
    const u64 ZEROING_TIME = 100 * FREQ_MHZ; // 100us

    // `have_free_hpage` checks the allocator free list.
    enum free_hpage_status fhps = have_free_hpage();
    u64 alloc_cost = fhps > fhps_none ? 0 : EXPENSIVE;
    u64 prep_cost = fhps > fhps_free ? 0 : ZEROING_TIME;
    cost->cost = alloc_cost + prep_cost;

    // BENEFIT = averted TLB miss cycles from profile.
    cost->benefit = compute_hpage_benefit(addr);
}
```

Figure 4.4: CBMM huge page cost-benefit model.

Profiling Our methodology generates for each workload a mapping from virtual memory regions (i.e., ranges of virtual addresses) to the number of averted usermode page walk cycles when the region is backed by huge pages. We modify the Linux kernel to give precise control over promotions. We then repeatedly run a given workload varying the set of promoted pages. We additionally run the workload with no huge pages as a baseline. We use hardware performance counters to measure the number of TLB misses, the number of cycles spent in pages walks, and the overall cycle count for kernelspace and userspace execution. We then take the difference in overhead and overall runtime between any given set of pages and the baseline. The size of the sets of promoted pages can be varied to tradeoff profiling time with precision. Our prototype uses the offset into allocation zones instead of virtual addresses, so that profiles tolerate Address Space Layout Randomization.

Broadly, we found that our workloads could be categorized as *high-processing* or *low-processing*. *High-processing* workloads, such as `xz`, `canneal`, or `mcf`, heavily process their input data to produce internal data structures; their memory access patterns are driven by computation over these data structures. *Low-processing*

workloads, such as `memcached`, `mongodb`, and `dc-mix` (see Section 4.4), often do little more than data storage and retrieval, so their access patterns are driven by client request patterns. We found that we can reliably distinguish between high- and low-processing workloads using the skewness [1] of the distribution of averted page walk cycles. High-processing workloads often have a small number of highly-impactful memory regions, so they have a high positive skew ($skew > 2$ seems to work empirically). When generating profiles for low-processing workloads, we assigned all regions a benefit equal to the mean benefit measured empirically. For high-processing workloads, we assigned each region the benefit it individually demonstrated.

At runtime, we can supply a profile to the kernel in the form of a CSV file that lists virtual address ranges and their benefit from huge pages. Our implementation aims to demonstrate the potential of our approach while remaining simple to implement. We do not attempt to account for phases in workload behavior, but our design is amenable to such an upgrade in the future by repeating the profiling process at multiple points during the workload's execution. We assume the workload size is stable but can handle other input changes; in Section 4.4, we use randomized inputs for most workloads.

Asynchronous Prezeroing

Background We examine *asynchronous prezeroing* as a means of improving the latency of large physical memory allocations. Asynchronous prezeroing clears free pages using a background daemon to save time during a page fault when it would slow down userspace programs. Our analysis indicates that prezeroing would reduce the cost of a huge page by almost two orders of magnitude.

There is much prior work on asynchronous prezeroing of pages [6, 56, 57, 89, 113, 135]. Prezeroing fell out of favor because the primary cost of zeroing 4KB pages is cache misses, but prezeroing pages leaves them cold when users access them, so latency is merely shifted to userspace [6, 135]. Recently, Panwar et al. find prezeroing is beneficial for huge pages and use non-temporal store instructions to avoid cache pollution [113]. However, their approach requires hand-tuning to avoid excessive CPU usage or lock contention on the page allocator. CBMM adapts their prezeroing implementation with a model to determine when and how long to run, eliminating the need for hand-tuning.

Model Listing 4.5 shows CBMM's model for running the asynchronous prezeroing daemon. The model makes numerous assumptions; most importantly, it assumes that CPU time is free unless taken away from userspace (i.e., the system is not idle) and that the chief costs of prezeroing are the execution time of zeroing and contention on the allocator lock, rather than cache pollution. This matches our own analysis and observations while working on CBMM. The chief benefit of prezeroing is to move zeroing overhead out of

[1] *skewness* is a statistical measure of distribution asymmetry.

the critical path of huge page promotion. For simplicity, we assume a constant processor frequency over short time windows, even though the frequency varies.

Also, this model exemplifies CBMM's iterative approach to building models. We started with a model that only accounted for CPU time and potential huge page allocations. As we ran experiments, we discovered the lock contention and improved the model to account for it by adding the lines labeled as `COST of lock contention` in Listing 4.5, resolving the performance issue. The entire process took less than a day of debugging, measurement, and implementation.

```
void prezeroing_model(mm_action *action, mm_cost_delta *cost)
{
    // COST of the runtime itself... Assume:
    // - Don't care about NUMA nodes.
    // - Zeroing costs ~10^6 cycles.
    __kernel_ulong_t cpu_load = get_avenrun();
    int ncpus = num_online_cpus();

    const u64 HPAGE_ZERO_COST = 1000000; // cycles

    // ncpus > cpu load average => idle cpu, free to run.
    if (ncpus > cpu_load) {
        cost->cost = 0;
    } else {
        cost->cost = HPAGE_ZERO_COST * action->prezero_n;
    }

    // COST of lock contention. Assume:
    // - Cost of lock acquisition = ~150cyc, do it 2x.
    // - Lock is unheld for ~1ms/horizon => free locking
    const u64 UNHELD = FREQ_MHZ * 1000; // = 1ms in cycles
    const u64 SINGLE_CS = 150; // cycles

    const u64 crit_sect_cost = SINGLE_CS * 2; // cycles
    const u64 nfree = UNHELD / crit_sect_cost;
    const u64 ntodo = action->prezero_n > nfree ? action->prezero_n - nfree : 0;
    cost->cost += ntodo * critical_section_cost;

    // BENEFIT. Assume past usage predicts future usage.
    u64 recent_used = mm_estimated_prezeroed_used();
    cost->benefit = min(action->prezero_n, recent_used) * HPAGE_ZERO_COST;
}
```

Figure 4.5: CBMM async prezeroing cost-benefit model

Eager paging

Background Eager paging allocates physical memory upon user request, rather than lazily on a page fault (the default) [83]. It enables large contiguous physical memory allocations, which are easier to back with huge pages and enable useful hardware optimizations [83,106,119,131,142]. However, a drawback to eager paging is memory bloat if the workload does not use all the allocated memory [83]. Preloaded profiles unlock this optimization while avoiding memory bloat.

Model Listing 4.6 shows CBMM's model for eager paging, which is invoked by `mmap` or `brk` system calls. It uses a preloaded profile to determine which subregions will be touched and assumes that the model has

Workload	Description	Input	Peak Mem
xz	data compression [1]	profiling: native input, eval: custom scaled up input	150GB
mcf	combinatorial optimization, scheduling [1]	native input	3GB
canneal	simulated annealing, chip routing [30]	custom input, randomly generated each time	150GB
mongodb	KV store	YCSB driver [39], 75%W-25%R	150GB
memcached	in-memory KV store	YCSB driver [39], 1%W-99%R	150GB
dc-mix	redis (KV store), memhog (microbench., creates fragmentation), metis (in-memory map-reduce) [82]	redis: YCSB driver [39], 50%W-50%R; memhog: N/A; metis: built-in	165GB

Table 4.1: Description of Workloads – their behavior, inputs, and peak memory usage.

perfect knowledge, allowing it to ignore the cost of potential bloat. If more than one page is being eagerly allocated, we create opportunity for contiguous allocation.

```
void eager_paging_model(vm_area_struct *mmap_region, mm_cost_delta *cost)
{
    // ASSUME: past usage predicts future; use profile.
    // COST: time to create new page.
    const u64 PF_NEW_PAGE = FREQ_MHZ * 10; // 10us in cycles

    struct range *ranges = prev_touched(mmap_region);
    cost->cost = len(ranges) * PF_NEW_PAGE;

    // BENEFIT: time to create new page, coalesced faults
    const u64 PF_CS = 300; // cycles
    cost->benefit = len(ranges) * PF_NEW_PAGE + (len(ranges) - 1) * PF_CS;
}
```

Figure 4.6: CBMM eager paging cost-benefit model

Profiling We profile eager paging behavior by periodically reading the `/proc/<pid>/pagemap` file while the workload is running. This file contains information about memory mappings for the given process and allows us to detect which virtual memory regions have been faulted in. Pages that were faulted in during the execution are noted in the profile, and the model assumes they will be faulted in again in the future.

4.4 Evaluation

CBMM seeks to improve consistency while matching or exceeding the performance and efficiency of existing systems. We evaluate CBMM along multiple axes. First, we evaluate the page fault latency of CBMM to understand its consistency compared to Linux and HawkEye. Second, we measure the end-to-end performance of CBMM. Third, we look at the efficiency of CBMM's use of huge pages. Finally, we evaluate the generality of our approach by looking at the sensitivity of performance to profile changes.

Methodology

Table 4.1 describes our workloads. They represent a variety of software behaviors and exercise the kernel in different ways. `mongodb`, `memcached`, and `dc-mix` are memory-intensive workloads common in datacenters. `mongodb` and `memcached` are data stores, and `mongodb` is I/O heavy and makes use of the page cache. `dc-mix`

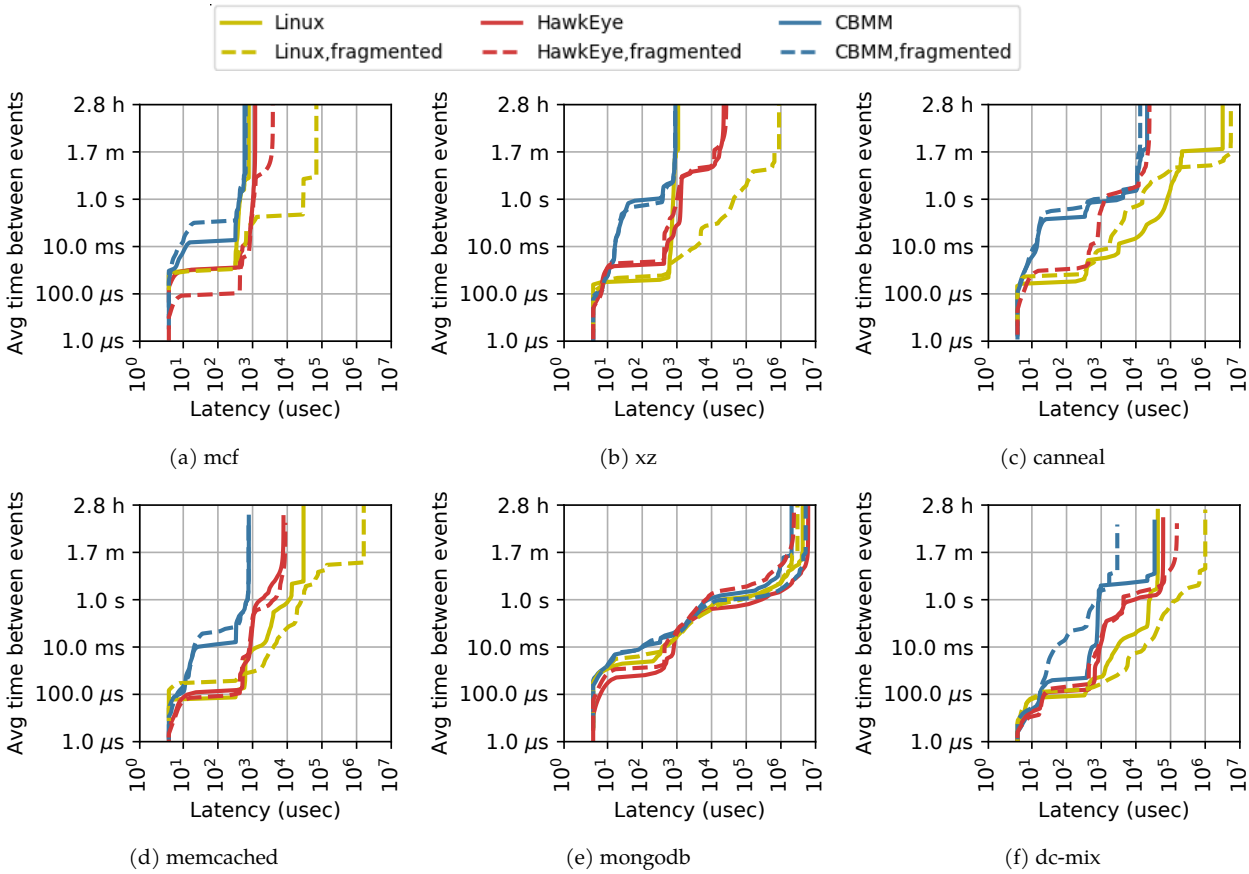
induces memory pressure and tries to simulate a real system in which a server, device driver, and batch job are using system resources. We drive the data stores in these workloads using YCSB [39] with different read-write ratios to increase the variety of MM behavior. `mcf`, `xz`, and `canneal` are computational workloads. We scale up the inputs of `xz` and `canneal` to use more memory. In all experiments with server applications (e.g., memcached, redis, mongodb), we run the client program on the same machine as the server, so as not to measure network effects. We run each workload with its default number of threads and pin all workloads to one NUMA node to reduce variation caused by NUMA effects. To reduce noise, we run each experiment 5 times and report the median results. For all workloads except `mcf` and `xz`, the input is randomized and changes between executions of the workload. For `xz`, we use the native input to generate a profile and use a custom input when evaluating performance.

All experiments run on CloudLab [122] c220g5 machines with two Intel Xeon Silver 4114 (10C/20T, 2.2 GHz, Skylake 2017), 192GB 2666MHz DDR4 ECC DRAM, and a 480GB SAS SSD. We set the CPU scaling governor to `performance`. *Unless otherwise noted, we do not tune our systems at all*; the results represent CBMM's "out of the box" behavior.

We replace the system allocator with `jemalloc`, which is better in a datacenter setting and is used by Facebook [60]. All experiments run on CentOS 7.8.2003 with the relevant kernel. We disable Meltdown and Spectre [86, 93] mitigations, which cause severe performance degradation. We use unmodified Linux 5.5.8 with Transparent Huge Pages enabled as our baseline. We configure CBMM similarly to Linux but we preload a profile of huge page benefits and eager paging, as derived in Sections 4.3 and 4.3. We also compare against HawkEye [113], a state-of-the-art research huge page management system based on Linux 4.3. We configure HawkEye as in its paper, including its prezeroing daemon. We ran experiments against stock Linux 4.3 and found that it performs within 15% of Linux 5.5.8 on average (see Figure 4.8). To measure page fault latency in HawkEye, which runs on a different kernel without our instrumentation, we use eBPF to instrument the `handle_mm_fault` function, which represents the main portion of the page fault handler. We found that `canneal` crashes with a segfault on HawkEye when the system is unfragmented, so we omit that experiment from results.

Fragmentation We run each workload on a freshly rebooted system and on a preconditioned system. Preconditioning aims to simulate a long-running datacenter environment by inducing external fragmentation, which hinders large physical memory allocations, such as huge pages.

We had difficulty identifying a reproducible fragmentation methodology. We attempted to reuse techniques from prior work [113, 142, 144] and also made several attempts at our own methodologies with little success; on Linux, deferred freeing of physical memory and kernel daemons such as kcompactd and kswapd



(a) mcf (b) xz (c) canneal

(d) memcached (e) mongodb (f) dc-mix

Figure 4.7: Soft page fault tail latency distribution on each system, weighted by page fault rate. A point (x, y) on the plot indicates that a fault of latency $\geqslant x$ happens at an interval of $\geqslant y$ on average.

cause variable results. Also, each methodology preconditions machines in a different way, none of which is obviously more realistic than the others.

For our evaluation, we choose a simple methodology derived from prior work [51, 142, 144]. We enable `CONFIG_SLAB_FREELIST_RANDOM` and `CONFIG_SHUFFLE_PAGE_ALLOCATOR` when compiling the kernel and add a sysfs file that triggers shuffling of the kernel physical memory free lists. To precondition the system, we reboot and then trigger free list shuffling. Then we run a program that allocates all system memory (with `mmap(MAP_POPULATE)`) and frees all but the first page of each 2MB region before sleeping for the duration of the workload. This methodology is simple and yields comparable results to other methodologies. We measure the Free Memory Fragmentation Index [68, 87, 113, 144] after preconditioning but before the workload begins. On CBMM and HawkEye, preconditioning consistently leaves around 183GB of free memory with 99% fragmentation. On Linux, half of runs experience similar results, but in the other half, deferred page freeing causes $<$ 2GB of memory to be considered free, making it difficult to measure fragmentation.

System Behavioral Consistency

Figure 4.7 shows tail latency on each kernel without fragmentation (when latency should be lowest); note the log x- and y-scales. To account for differences in the frequency of page faults due to differing MM decisions, we show the average interval between events, rather than the percentile on the y-axis.

Unlike Linux (Figure 4.2k), CBMM rarely attempts an expensive fallback path (e.g., compaction or reclamation) during huge page promotion, even under fragmentation; allocation failures usually result in the allocation of base pages. CBMM often experiences more page faults than Linux or HawkEye, but as Figure 4.7 shows, CBMM still sees a lower rate of long page faults than they do because its cost-awareness leads to fewer pathological cases, falling back to 4KB pages instead.

Even without fragmentation, CBMM always matches or improves on the tail latency of Linux and HawkEye, often by wide margins. In `xz`, `canneal`, and `memcached`, CBMM reduces the frequency of page faults taking 10-1000µs by two to three orders of magnitude compared to Linux or HawkEye. In `canneal` and `memcached`, CBMM reduces the frequency of (or eliminates) all page faults slower than 10µs by two or more orders of magnitude compared to Linux. In `memcached`, page faults taking over 1ms are nearly eliminated, while in `dc-mix`, they are reduced in frequency from nearly constant in Linux to every 10s or longer in CBMM. `mongodb` uses a userspace asynchronous I/O framework, as previously discussed, so its page fault latencies are dominated by context switches and other userspace threads; thus, our improvements are not visible in the figure. However, the figure does show that CBMM does not regress page fault latency, and as we will see in the next section, CBMM achieves significantly better performance than Linux or HawkEye for this workload.

Under fragmentation, CBMM usually achieves even larger tail latency improvements, particularly compared to Linux. For all workloads except `mongodb`, CBMM reduces the frequency of all page faults taking $\geqslant$ 50µs by 1-3 orders of magnitude compared to Linux and up to one order of magnitude compared to HawkEye. `mongodb` performs similarly to the unfragmented case, as discussed above.

Summary CBMM improves tail latency compared to Linux or HawkEye. For multiple workloads, CBMM reduces the frequency of slow page faults by one or more orders of magnitude, especially under fragmentation.

End-to-End Performance

CBMM's major goal is to improve consistency and the debuggability of MM-related performance issues without degrading performance. Figure 4.8 shows the performance of each kernel with and without fragmentation. All results are normalized to Linux *without* fragmentation. Note that some of the performance difference of HawkEye compared to the other systems is due to Linux 4.3 (the black bar in the figure). On

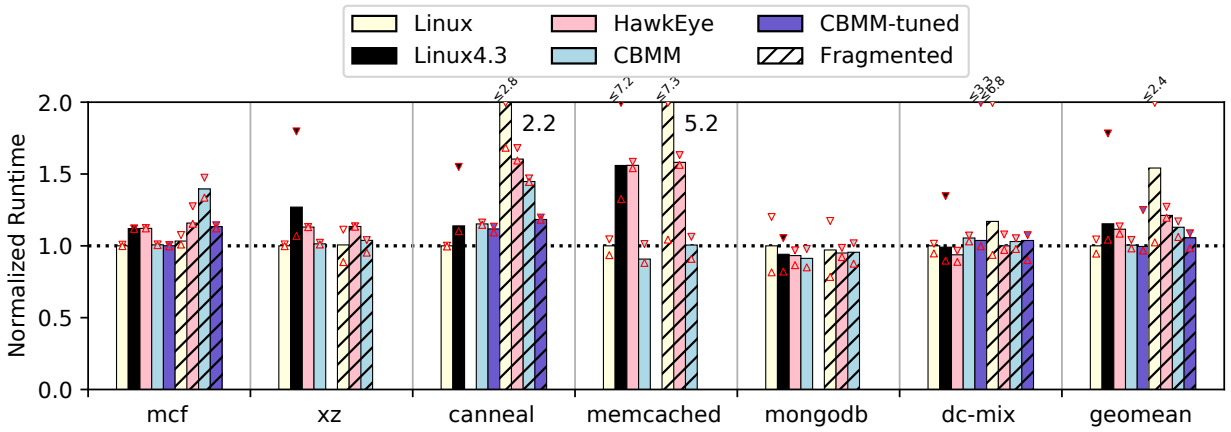


Figure 4.8: Runtime of workloads on each kernel, normalized to Linux with THP without fragmentation (lower is better). The upper and lower triangles around each bar represent the max and min observed value. The "$\leqslant$ MAX" text at the top of the plot indicates max values above the boundary of the plot.

average, *without* fragmentation, CBMM has performance comparable to Linux and better than HawkEye. On average, *with* fragmentation, *CBMM is 7% and 30% faster than HawkEye and Linux*; in fact, it is only 12% slower than without fragmentation. With minimal tuning, on average, CBMM is *13% and 35% faster than HawkEye and Linux* under fragmentation.

Without fragmentation, CBMM matches or exceeds the performance of Linux or HawkEye for all workloads except `canneal`. For `canneal`, CBMM is 15% slower than Linux because our profiles underestimate the benefit of huge pages. For `mongodb`, CBMM is 9% faster than Linux because it uses significantly more huge pages.

With fragmentation, CBMM outperforms Linux and/or HawkEye for all workloads except `mcf`. `mcf` uses too little memory to induce memory pressure; thus, CBMM overestimates the cost of huge pages and uses significantly fewer huge pages than Linux. In all other workloads, CBMM matches or outperforms at least one of Linux or HawkEye, often by wide margins. In `dc-mix`, `canneal`, and `memcached`, CBMM outperforms Linux by 34%, 34% and 81%, respectively, because its cost models allow it to adapt to a fragmented context, reflecting CBMM's focus on consistent behavior. Notably, this includes all of our datacenter workloads.

To demonstrate CBMM's benefit to performance debugging, we tune the performance of `mcf`, `canneal`, and `dc-mix` beyond the above results. In `mcf` and `dc-mix`, CBMM underestimates the benefit of huge pages, so we adjust the benefit upward in the respective profiles. We found that `canneal` exhibits a strong tradeoff between performance and page fault tail latency. As `canneal` is a non-interactive computational workload, we optimize for end-to-end performance by adjusting the profile to more aggressively allocate huge pages for the most import memory regions. After tuning, `dc-mix` without fragmentation runs 2% faster, and `mcf` with fragmentation runs 19% faster, than without tuning, but neither has a regression in tail latencies. `canneal`

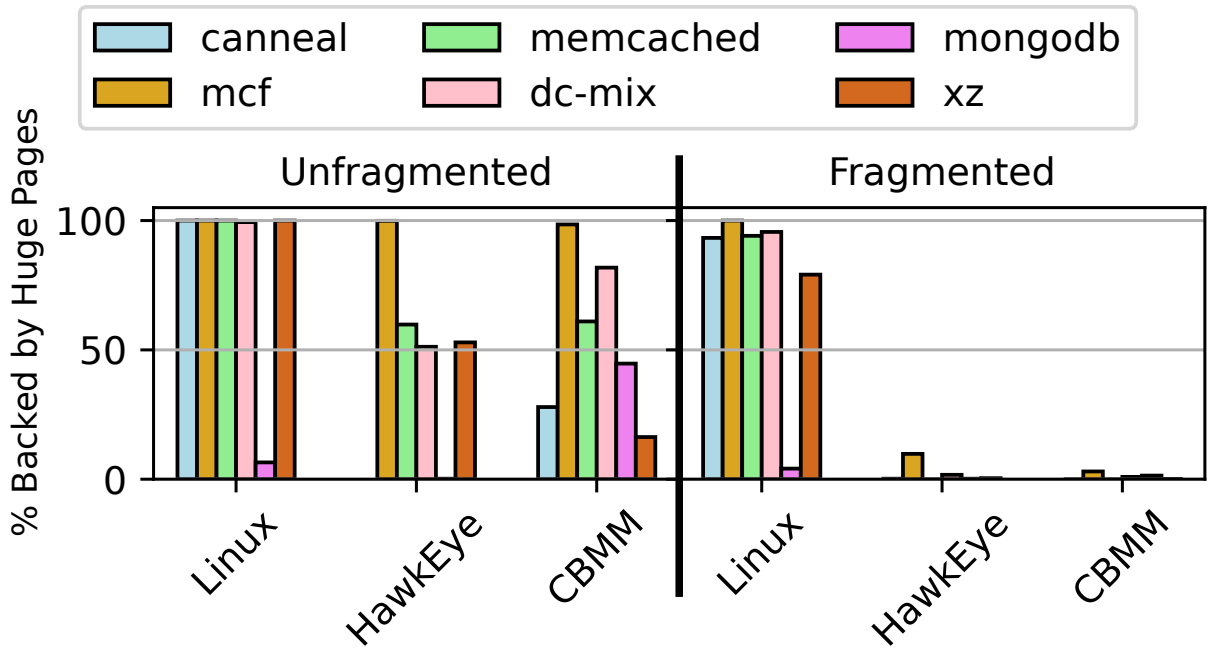


Figure 4.9: Percent of anonymous memory backed by huge pages on CBMM, HawkEye, and Linux with THP.

runs 18% faster than without tuning (46% faster than Linux) at the expense of some degradation in tail page fault latencies. In total, the tuning effort took less than a week, most of which was spent waiting for workloads to run.

Summary CBMM's has competitive performance with Linux/THP and HawkEye *and* better tail latency and more interpretable behavior. In most cases, CBMM matches or exceeds Linux's performance. Under fragmentation, CBMM often performs vastly better than Linux or HawkEye because of its focus on consistent behavior. Also, CBMM is easily debuggable and tunable by adjusting profiles and/or models.

Efficiency

Allocating huge pages to memory regions that do not need them wastes contiguous memory and promotion overheads and possibly bloats memory usage. Figure 4.9 shows the percentage of anonymous memory used by each workload that is backed by a huge page in CBMM, HawkEye, and Linux (with THP) with and without fragmentation preconditioning.

Generally, preloaded profiles drive CBMM's huge page usage, while HawkEye and Linux are more indiscriminate with huge page promotion. Usually, Linux attempts to use more huge pages than CBMM or HawkEye, often backing almost all memory with huge pages. HawkEye uses huge pages more efficiently than Linux, often achieving similar performance with much fewer huge pages. For most workloads, Linux

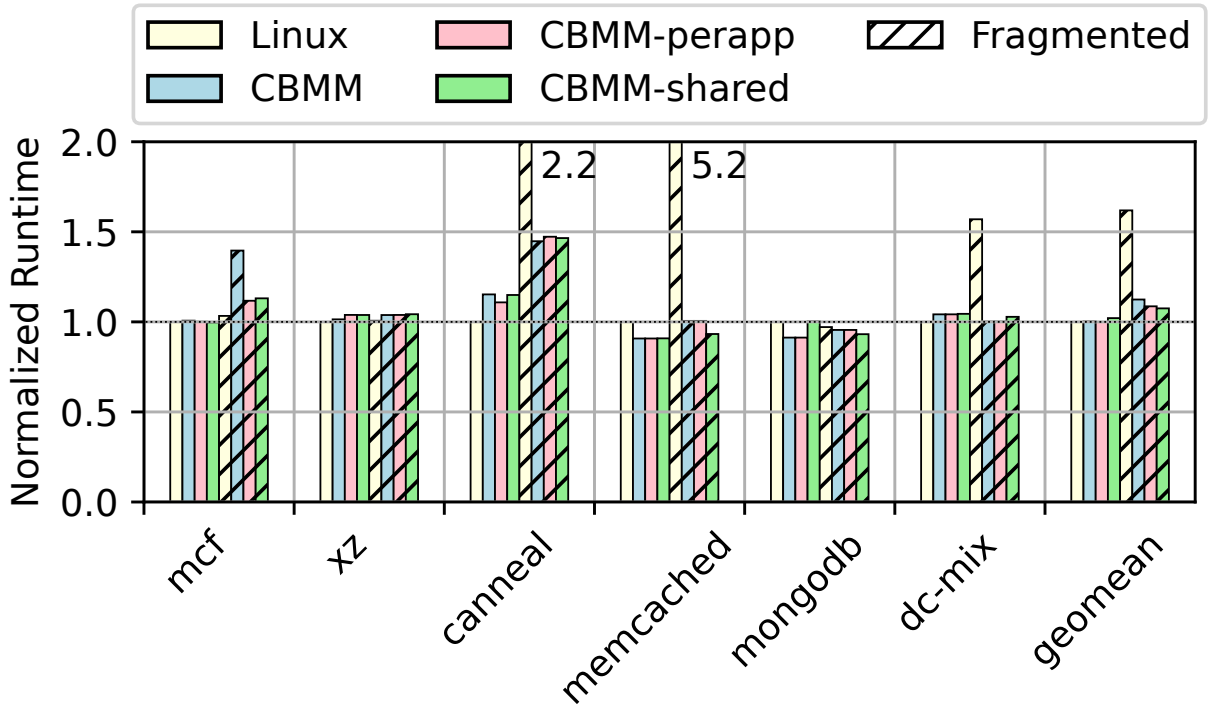


Figure 4.10: Runtime of CBMM workloads with generalized profiles, normalized to Linux with THP without fragmentations (lower is better).

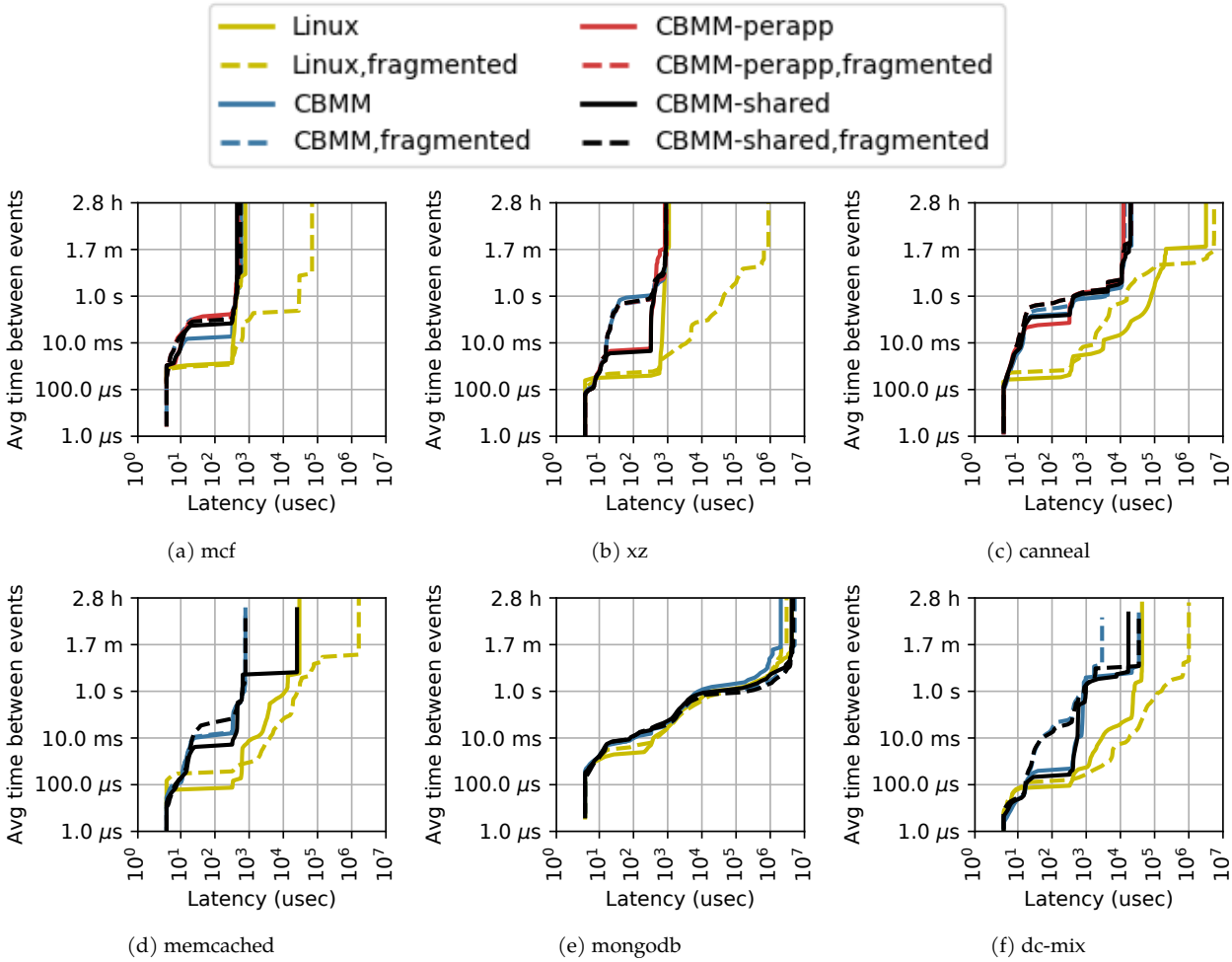
still attempts to use huge pages under fragmentation, whereas CBMM and HawkEye do not, leading to significantly better tail latencies, and often better performance.

For `xz`, CBMM's profile allows it to promote only a small but important part of the address space, so it matches Linux's performance (and outperforms HawkEye) while using almost 80% fewer huge pages. For `mongodb`, CBMM outperforms Linux and HawkEye by using more huge pages in the absence of fragmentation and fewer in its presence, exemplifying CBMM's cost-awareness.

Summary Despite having the most consistent behavior and sometimes better performance, CBMM often uses significantly fewer huge pages than Linux or HawkEye. By being cost- and context-aware, CBMM is more targeted in its use of huge pages, though in some cases, our profiles underestimate the benefit of huge pages.

Generality

CBMM has benefits even when a profile is highly imprecise, primarily by avoiding the pathological behavior of Linux. We compare three versions of profiles: the standard CBMM profile is as in Section 4.3, `perapp` assigns a single value to all memory regions in the workload equal to the average benefit of enabling THP for



(a) mcf (b) xz (c) canneal

(d) memcached (e) mongodb (f) dc-mix

Figure 4.11: Soft page fault tail latency distribution, weighted by page fault rate, for different profiles. Compare to Figure 4.7.

the workload, and `shared` is shared between all workloads and assigns a single value to all memory regions equal to the mean benefit of the `perapp` profiles.

Figure 4.11 shows how the different profiles affect page fault tail latency in `mcf` and `xz`. The `perapp` and `shared` profiles have minor regressions in page fault tail latencies compared to the standard profiles but still improve over Linux.

Figure 4.10 shows the how the different profiles affect performance. In most cases, *CBMM with the simpler profiles outperformed Linux* with fragmentation, and the performance differences between the three profiles are within 5%. The `perapp` and `shared` profiles outperform the standard profiles slightly in some workloads. One exception is `mcf` under fragmentation, where both the `perapp` and `shared` profiles outperform the standard profile by 20%, similar to the tuned profile in Section 4.4, by being more liberal with huge pages.

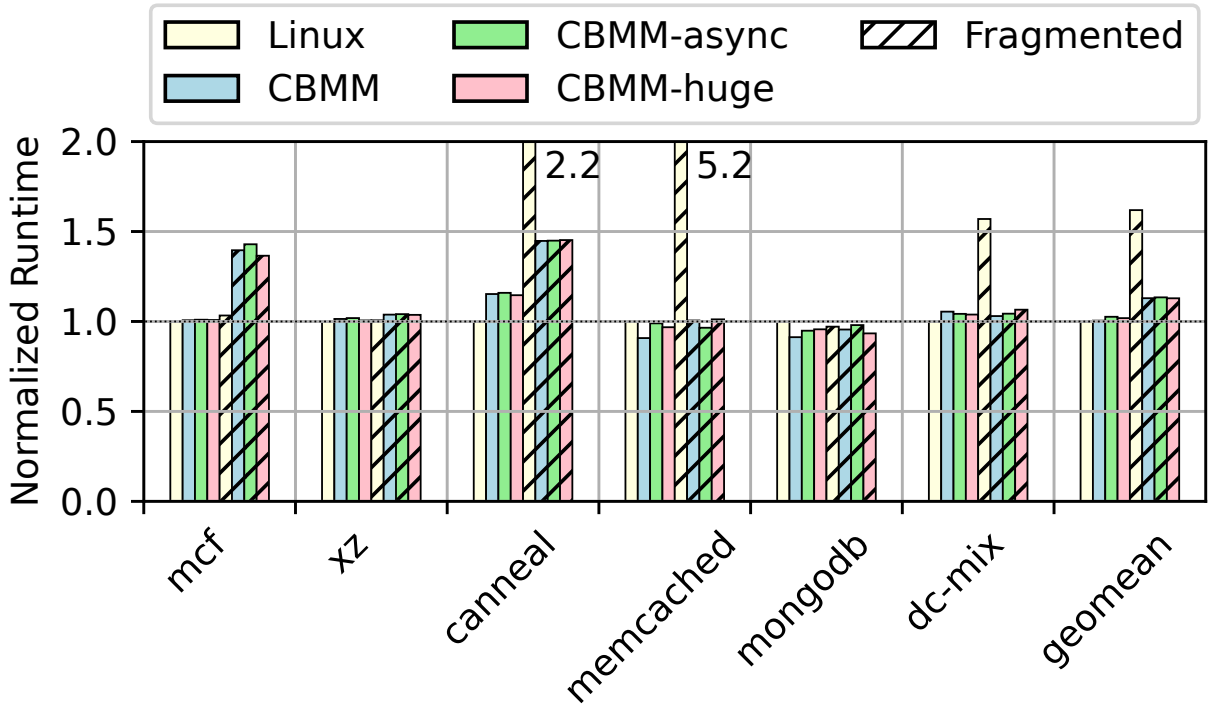


Figure 4.12: Runtime of CBMM workloads when enabling more models, normalized to Linux with THP without fragmentations (lower is better).

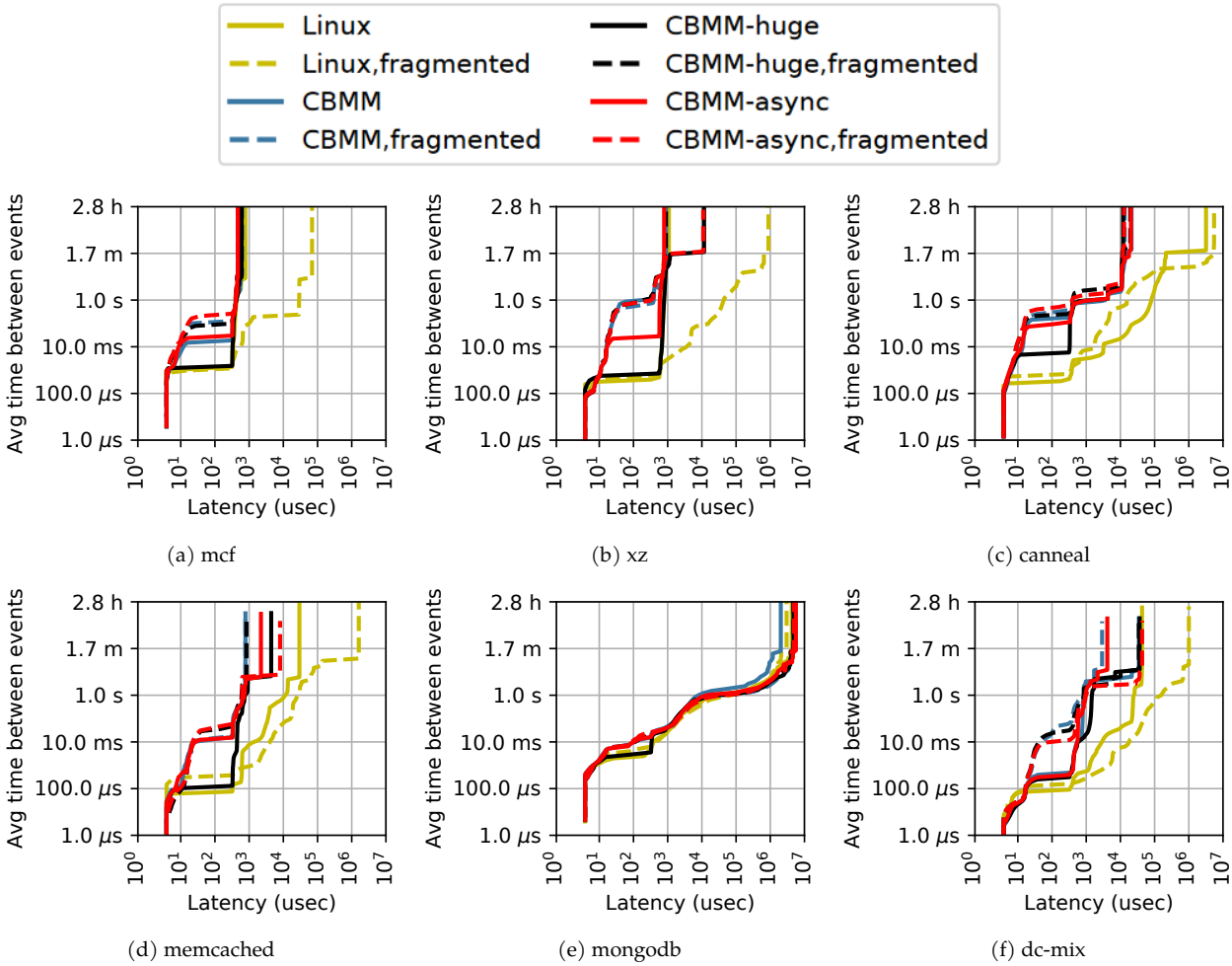
Summary More precise profiles improve CBMM's performance and tail latency, but imprecise profiles still have good results. Furthermore, profiles can be used to trade off performance and page fault latency.

CBMM Models

We evaluate the contribution of each model in Section 4.3 via three configurations of CBMM: `huge` enables only the huge page model, `async` additionally enables prezeroing, and standard CBMM enables all three models. Figure 4.12 shows the performance of these configurations, while Figure 4.13 shows tail latency for `mcf` and `xz`.

Each policy provides benefits in different settings. The huge page model alone (`huge`) captures most of the performance benefit of CBMM because it prevents costly huge page allocations.Asynchronous prezeroing (`async`) decreases page fault tail latency by making huge pages cheaper. It also reduces performance slightly on an unfragmented system, where free pages abound, because prezeroing is wasted work. With fragmentation, prezeroing has little effect on performance.

Eager paging does not directly benefit performance but enables larger contiguous allocations where hardware supports it [83, 106, 119, 131, 142]. To evaluate how well CBMM can make large allocations, we compare the number of eagerly allocated regions and peak memory usage of CBMM with eager paging



(a) mcf (b) xz (c) canneal

(d) memcached (e) mongodb (f) dc-mix

Figure 4.13: Soft page fault tail latency impact of different models. Compare to Figure 4.7.

against Linux with `MAP_POPULATE`, the `mmap` flag that eagerly maps memory. In all workloads, regardless of fragmentation, CBMM uses eager paging for nearly the entire working set of the workload. Thanks to profiles, CBMM has $< 1\%$ memory bloat – 3%-48% less memory than `MAP_POPULATE` would use.

Summary CBMM's huge page model provides significant tail latency (and often performance) improvements. Asynchronous prezeroing enables more huge page usage under fragmentation, but has a modest cost on unfragmented systems. Eager paging has a modest performance cost but enables more contiguous memory allocation.

4.5 Future Work

Our work demonstrates the importance of deeply understanding the behavior of existing systems; numerous aspects of the MM remain uncharacterized and poorly understood. For example, file caches and driver-used

memory can comprise significant portions of system memory usage, but little prior work has characterized them. It is also unclear how other kernel subsystems impact MM, e.g., block I/O, scheduler. Understanding these aspects of system behavior is key to making MM suitable for modern workloads.

In our work, we found that often existing instrumentation was insufficient for understanding system behavior. For example, while BPF is useful for measuring the latency of different events (e.g., different types of page faults), we found it unreliable. Various compiler optimizations such as code inlining and stack frame elision make it hard for BPF to attach to different events, while unstable interfaces break custom instrumentation between kernel versions. These issues highlight the need for better instrumentation, as well as for designing MM code with instrumentation as a first-class concern.

Finally, we believe that policy generation, management, and maintenance can be centralized in the cluster manager, allowing all machines in a cluster to take advantage of 1st-party datacenter workloads. We have done some initial exploration of this idea [97], but more work is needed to demonstrate the viability of such an approach.

4.6 Conclusion

Modern computing needs are placing new demands on kernel MM. To meet these demands, kernel MM must begin to prioritize behavioral consistency and debuggability. We propose CBMM, a MM system that uses cost-benefit analysis to make policy decisions. Despite using relatively simple models in its cost-benefit estimation, CBMM's principled approach to MM allows matching the performance of existing systems while also improving system behavioral consistency. CBMM paves a way for kernel MM behavior to become less opaque, unlocking further performance and optimizations in the future.

5 CHARACTERIZING PHYSICAL MEMORY FRAGMENTATION

A [system] [fragmented] against itself cannot [run memcached].

— Abraham Lincoln, 1858

Acknowledgements. *This chapter contains work co-authored with Michael M. Swift.*

We thank CSL and HTCondor for allowing us to instrument their systems; in particular, many thanks go to Michael Gibson (CSL) and Aaron Moate (HTCondor), who put significant time and effort into helping us plan, build, and deploy the instrumentation; we could never have done this work without you. We thank David Merrell for early insightful discussions about data analysis. We also thank Jing Liu, Sujay Yadalam, and Anthony Rebello, for their insightful feedback on drafts of this paper, and the anonymous reviewers for their feedback on our submission.

This research was performed using the compute resources and assistance of the UW-Madison Center For High Throughput Computing (CHTC) in the Department of Computer Sciences. The CHTC is supported by UW-Madison, the Advanced Computing Initiative, the Wisconsin Alumni Research Foundation, the Wisconsin Institutes for Discovery, and the National Science Foundation, and is an active member of the OSG Consortium, which is supported by the National Science Foundation and the U.S. Department of Energy's Office of Science.

Surprisingly little prior work exists to characterize or quantify fragmentation, even though many papers refer to its impact and many designs seek to mitigate or avoid it. For example, it is unclear what contiguous memory allocations are actually used for, the degree to which workloads fragment system memory, what types of memory allocations cause fragmentation, how fragmentation changes over time, etc. To fill the above gaps in knowledge, we conduct the first study of external physical memory fragmentation in productions settings.

5.1 Study: Linux Allocation Sizes

It is unknown what allocation sizes are common, and thus what causes and is impacted by fragmentation. We use BPF to record all contiguous multi-page allocations in Linux for several batch and server workloads on server-class machines using a similar experimental setup to CBMM.

Finding 1. *In Linux, the most common use of contiguous memory is huge pages (512 contiguous pages). I/O-heavy processes also use contiguous memory for buffers and slab allocators for kernel data structures (usually 2-8 contiguous pages). (Table 5.1)*

Process	Description	Contiguous Memory Usage (Size per Allocation in Pages)
`java`	Java virtual machine	THP (512), Slab(memory mapping, process) (2-8)
`jbd`	ext4 journaling kernel thread	Slab(block I/O) (2-8)
`lettuce-epol`	Asynchronous I/O (`redis`)	UDS (2-4), THP (512)
`mcf_s`	SPEC 2017 mcf	THP (512)
`memcached`	In-memory key-value store	THP (512)
`pgrep`	Searches for processes by name	Slab(file,dirent,path) (2-8)
`redis`	Redis key-value store server	UDS (2-4), THP (512), Slab(filesystem) (8)
`sshd`	Secure Shell daemon (server)	Networking packetization buffer (8), Slab(TCP, memory mapping) (2-8)
`sudo`	Execute as superuser	Slab(memory mapping, file, path) (2-8), THP (512)
`xz_s`	SPEC 2017 xz	THP (512), Slab(file,dirent,path) (2-8)

Table 5.1: Contiguous memory usage (in order of number of allocations) of notable processes in workloads. THP = Transparent Huge Pages. UDS = Unix Domain Socket buffer. Slab(X) = kernel slab allocator for X data structures.

Typically, a single process's use of contiguity was dominated by one use case. In `memcached`, `mcf`, and `xz`, huge pages comprise 84%, 48%, and 86% of all large allocations. In `redis`, almost 100% of large allocations back a Unix Domain Socket buffer for threads to communicate. In `sshd`, 87% of contiguous memory is for network data packetization buffers. All processes triggered kernel slab allocations for kernel data structures, which have been shown to increase fragmentation [115]. Daemons, such as ext4's journaling thread, also make slab allocations (e.g., I/O buffers).

Implications: Though Linux maintains intermediate-sized free lists, they are rarely used except to fill lower-order lists. Separating the allocation mechanisms for 2-8 page chunks and huge pages may preserve contiguity for longer. For example, smaller allocations could be made from a different part of the physical address space from huge pages.

5.2 Study: Real World Fragmentation

In order to gain a better understanding of fragmentation in the real world, we instrument live machines in the Computer Sciences department at University of Wisconsin-Madison, including departmental infrastructure and infrastructure and compute nodes from the Center for High-Throughput Computing.

Methodology

Host Type (Amt)	CPU Thds/Cores/Sockets (Family; N machines if >1)	Mem (N if >1)	Description
`hpc-exec` (8 BM)	48T/48C/4S (Opteron'13), 32T/16C/2S (Sandy Bridge'12), 40T/20C/2S (Skylake'17), 40T/20C/2S (Ivy Bridge'13), 4T/4C/1S (HarperBridge'08)	128GB (5), 512GB (2), 16GB (1)	Generic execution hosts

Host Type (Amt)	CPU Thds/Cores/Sockets (Family; N machines if >1)	Mem (N if >1)	Description
hpc-gpu1 (4 BM)	32T/16C/2S (Haswell'14), 40T/20C/2S (Skylake'17), 96T/48C/2S (EPYC'19)	2TB (2), 512GB, 128GB	GPU execution hosts
hpc-gpu2 (2 BM)	160T/80C/4S (Skylake'17), 40T/20C/2S (Skylake'17)	512GB, 256GB	Dockerized GPU workloads for particular project
hpc-hmem (2 BM)	80T/40C/4S (Ivy Bridge'14), 80T/40C/4S (Broadwell'16)	2TB, 4TB	High-memory hosts
hpc-hypr (6 BM)	32T/16C/2S (Sandy Bridge'12), 12T/12C/2S (Opteron'10)	96GB (4), 64GB (1), 32GB (1)	Hypervisor hosts
hpc-ceph (2 BM)	32T/16C/2S (Ivy Bridge'13)	128GB	Ceph object store hosts
hpc-subm (2 BM)	32T/16C/2S (Sandy Bridge'12)	96GB	User access points for submitting jobs
hpc-es (1 BM)	12T/12C/2S (Opteron'10)	64GB	Host running elastic search + kibana
hpc-graf (1 BM)	48T/48C/4S (Opteron'13)	128GB	Host running graphana + nosql database
hpc-fprx (2 VM)	4 vCPUs (Intel, AMD)	32GB	Squid forward proxy
hpc-rprx (2 VM)	12 vCPUs (Intel)	32GB	Squid reverse proxy
hpc-stag (1 VM)	4 vCPUs (AMD)	8GB	Host for staging data into local HPC storage
hpc-web (1 VM)	4 vCPUs (Intel)	4GB	Static content web server
hpc-dkca (1 VM)	2 vCPUs (Intel)	4GB	Docker caching registry
hpc-gang (1 VM)	8 vCPUs (AMD)	64GB	Ganglia time-series metric gatherer
hpc-mntr (1 VM)	4 vCPUs (Intel)	8GB	Icigna/nagios monitoring service
cs-rsch (32 BM, 12 VM)	2 vCPUs (Broadwell '16; 7), 4T/4C/1S (Yorkfield '08; 6), 56T/28C/2S (Broadwell '16; 3), 2T/2C/1S (Blackford '05; 3), 1 vCPU (Broadwell '16; 3), 4T/4C/1S (Yorkfield '08; 2), 32T/32C/4S (Opteron '13; 2), 128T/64C/2S (EPYC '17), 128T/64C/2S (EPYC '20), 48T/24C/2S (Cascade Lake '19), 48T/24C/1S (EPYC '20), 40T/20C/2S (Skylake '17), 32T/32C/2S (Opteron '13), 32T/16C/2S (Skylake '17), 32T/16C/2S (Sandy Bridge '12), 32T/16C/2S (Haswell '14), 32T/16C/2S (EPYC '19), 24T/12C/1S (Ryzen '19), 16T/16C/2S (Opteron '13), 4T/4C/1S (Opteron '10), 4T/4C/1S (Ivy Bridge '12), 2C/2T/1S (Opteron '09), 15 vCPUs (Broadwell '16), 2 vCPUs (Haswell '14)	2.3TB (1), 1TB (1), 512GB (3), 384GB (1), 256GB (5), 225GB (1), 64GB (4), 33GB (1), 32GB (5), 16GB (1) 8GB (12), 4GB (3), 2GB (6)	Various hosts used by different research projects

Host Type (Amt)	CPU Thds/Cores/Sockets (Family; N machines if >1)	Mem (N if >1)	Description
cs-misc (23 VM, 6 BM)	1 vCPU (Broadwell '16; 8), 2 vCPUs (Broadwell '16; 7), 4 vCPUs (Broadwell '16; 5), 1 vCPU (Haswell '14; 3), 20T/10C/1S (Cascade Lake '19; 3), 64T/32C/2S (Cascade Lake '19), 2T/2C/1S (Blackford '05), 4T/4C/1S (Yorkfield '08)	192GB (1), 64GB (3), 32GB (1), 16GB (3), 8GB (5) 4GB (6), 2GB (10)	Misc hosts supporting specific use cases/projects
cs-cuda (16 BM)	48T/24C/1S (EPYC '19; 4), 48T/24C/2S (Broadwell '16; 3), 64T/32C/2S (EPYC '19; 2), 64T/32C/2S (Cascade Lake '19; 2), 48T/24C/2S (Cascade Lake '20), 96T/48C/2S (Cascade Lake '20), 48T/24C/2S (Cascade Lake '19), 36T/18C/1S (Skylake '17), 56T/28C/2S (Broadwell '16)	512GB (4), 480GB (1), 256GB (5), 192GB (2), 128GB (3), 96GB (1)	Research hosts using Nvidia CUDA [107]
cs-slrm (15 BM)	48T/24C/2S (Haswell '14; 13), 32T/16C/2S (Skylake '17), 24T/12C/1S (Haswell '14)	384GB (1), 128GB (14)	SLURM compute nodes run on behalf of Stats dept
cs-doop (3 BM)	48T/24C/2S (Cascade Lake '19; 3)	64GB	Hadoop cluster used for research
cs-igpu (4 BM)	64T/32C/2S (Cascade Lake '19; 4)	512GB	Instructional lab machines with GPUs
cs-afs1 (11 BM)	16T/8C/1S (Cascade Lake '20; 4), 8T/4C/1S (Broadwell '16; 3), 4T/4C/1S (Opteron '10; 2), 20T/10C/1S (Cascade Lake '19), 4T/2C/1S (Clarkdale '10)	64GB (1), 32GB (4), 16GB (3), 4GB (3)	AFS servers [74, 110]
cs-afs2 (6 VM)	1 vCPU (Broadwell '16; 3), 1 vCPU (Haswell '14; 3)	2GB (3), 1GB (3)	AFS servers [74, 110]
cs-sush (6 VM)	1 vCPU (Broadwell '16; 5), 4 vCPUs (Broadwell '16)	16GB (1), 2GB (5)	Backup servers
cs-msql (4 VM)	1 vCPU (Broadwell '16; 3), 2 vCPUs (Haswell '14)	2GB	MySQL databases
cs-nfs (3 VM, 1 BM)	1 vCPU (Broadwell '16; 2), 2 vCPUs (Broadwell '16), 48T/24C/2S (EPYC '19)	128GB (1), 2GB (3)	NFS servers
cs-pstg (2 VM)	2 vCPUs (Broadwell '16; 2)	4GB, 2GB	PostgreSQL servers
cs-dns (6 VM, 1 BM)	2 vCPUs (Broadwell '16; 3), 1 vCPU (Broadwell '16; 3), 4T/4C/1S (Lynnfield '09)	8GB (1), 4GB (2), 2GB (3), 1GB (1)	DNS servers, recursers, load balancers
cs-mntr (3 VM)	4 vCPUs (Broadwell '16; 3)	16GB (1), 4GB (2)	Network/application monitoring
cs-vaul (3 VM)	2 vCPUs (Broadwell '16; 3)	2GB	HashiCorp Vault secret management servers
cs-prx (3 VM)	2 vCPUs (Broadwell '16; 2), 1 vCPU (Broadwell '16)	4GB (1), 2GB (2)	Various proxy servers (Haproxy, squid, REST)
cs-dhcp (2 VM)	2 vCPUs (Broadwell '16; 2)	2GB	DHCP hosts

Host Type (Amt)	CPU Thds/Cores/Sockets (Family; N machines if >1)	Mem (N if >1)	Description
`cs-conf` (2 VM)	16 vCPUs (Broadwell '16), 4 vCPUs (Broadwell '16)	32GB, 8GB	Configuration management servers
`cs-vpn` (1 VM)	2 vCPUs (Broadwell '16)	2GB	VPN server
`cs-tftp` (1 VM)	2 vCPUs (Broadwell '16)	2GB	TFTPD server used for installs
`cs-apt` (1 VM)	2 vCPUs (Broadwell '16)	2GB	Apt-Cacher Debian package cache
`cs-web` (22 VM)	2 vCPUs (Broadwell '16; 15), 1 vCPU (Broadwell '16; 4), 2 vCPUs (Haswell '14; 3)	4GB (2), 2GB (19), 1GB (1)	Web servers for dept/faculty/staff/student web sites
`cs-ldap` (7 VM)	2 vCPUs (Broadwell '16; 3), 4 vCPUs (Broadwell '16; 2), 1 vCPU (Broadwell '16; 2)	8GB, 4GB (4), 2GB (2)	LDAP hosts
`cs-cweb` (5 VM)	4 vCPUs (Broadwell '16; 3), 2 vCPUs (Broadwell '16; 2)	16GB, 8GB, 4GB, 2GB (2)	Containerized web applications
`cs-mail` (4 VM)	2 vCPUs (Broadwell '16; 4)	4GB (3), 2GB (1)	Mail servers
`cs-gitl` (3 VM)	8 vCPUs (Broadwell '16), 4 vCPUs (Broadwell '16), 2 vCPUs (Broadwell '16)	12GB, 4GB, 2GB	Gitlab server
`cs-tckt` (1 VM)	4 vCPUs (Broadwell '16)	8GB	Request Tracker 4 ticket tracker
`cs-cal` (1 VM)	1 vCPU (Broadwell '16)	2GB	Department events calendar server
`cs-rsyn` (1 VM)	2 vCPUs (Broadwell '16)	2GB	rsync server for external collaborators

Table 5.2: Description and count of instrumented nodes. `cs-*` are department infrastructure hosts; `hpc-*` are high-performance computing infrastructure. "BM/VM" indicates whether the node is bare-metal or a virtual machine.

We observed 248 instrumented machines total, from X's department infrastructure (DI) and high-performance computing (HPC) infrastructure, spanning several generations and manufacturers of x86 hardware (Intel Blackford '05 and AMD Opteron '09 to AMD EPYC'19 and Intel Cascade Lake '20), and different sizes of memory (from 1GB to 4TB). We observed 45 different classes of workloads (as classified by system administrators; e.g., file servers, web servers, hypervisors, research workstations, and compute nodes from a high-performance computing cluster), detailed in Table 5.2. Many DI nodes are virtual machines, whereas most HPC nodes are bare-metal hardware. DI nodes mostly run Ubuntu 20.04 with kernel 5.4; HPC nodes run Centos7 with kernel 3.10.0 or 5.0.8.

We instrumented all systems for 7 days from April 6 to April 13, 2022. In every 30-minute window during this period, we pick a random minute to perform a "snapshot" of the system. Our instrumentation captures the contents of `/proc/kpageflags`, which contains information about each physical 4KB page of memory

and is our primary view into the memory fragmentation of the system. We also capture the kernel version, hardware details, and uptime. For VMs, we capture all information at the guest OS level. No machines were captured at both the guest and host OS level.

Free Memory

We examine the amount of free memory on all machines to better understand the impact of fragmentation.

Finding 2. *Small VMs experience high memory pressure and churn. Machines with large amounts of memory often have lower fragmentation.*

DI nodes tend to use small VMs ($\leqslant$ 4GB total) with $\leqslant$ 20% free memory. Most DI workloads and several HPC workloads have less than 2GB of free memory at least 75% of the time. These small VMs are sized to have minimal idle memory (e.g., by using balloon drivers). Background tasks, such as logging or virus scanning, have high file cache usage that leads to churn (i.e., pages being reclaimed and recycled) even if load is low.

Machines with more memory are more likely to be either highly idle (i.e., most cores idle; $\geqslant$ 50% free memory) or highly utilized ($\leqslant$ 10% free memory). Often large amounts of memory are freed after a process terminates; as we will see later, this leads to lower fragmentation.

Implications: On small VMs, kernels may want to use more passive fragmentation control, such as careful page (re)placement, because memory pressure is high and resources are limited. On large machines with more resources, more aggressive fragmentation control (e.g., compaction) may be acceptable.

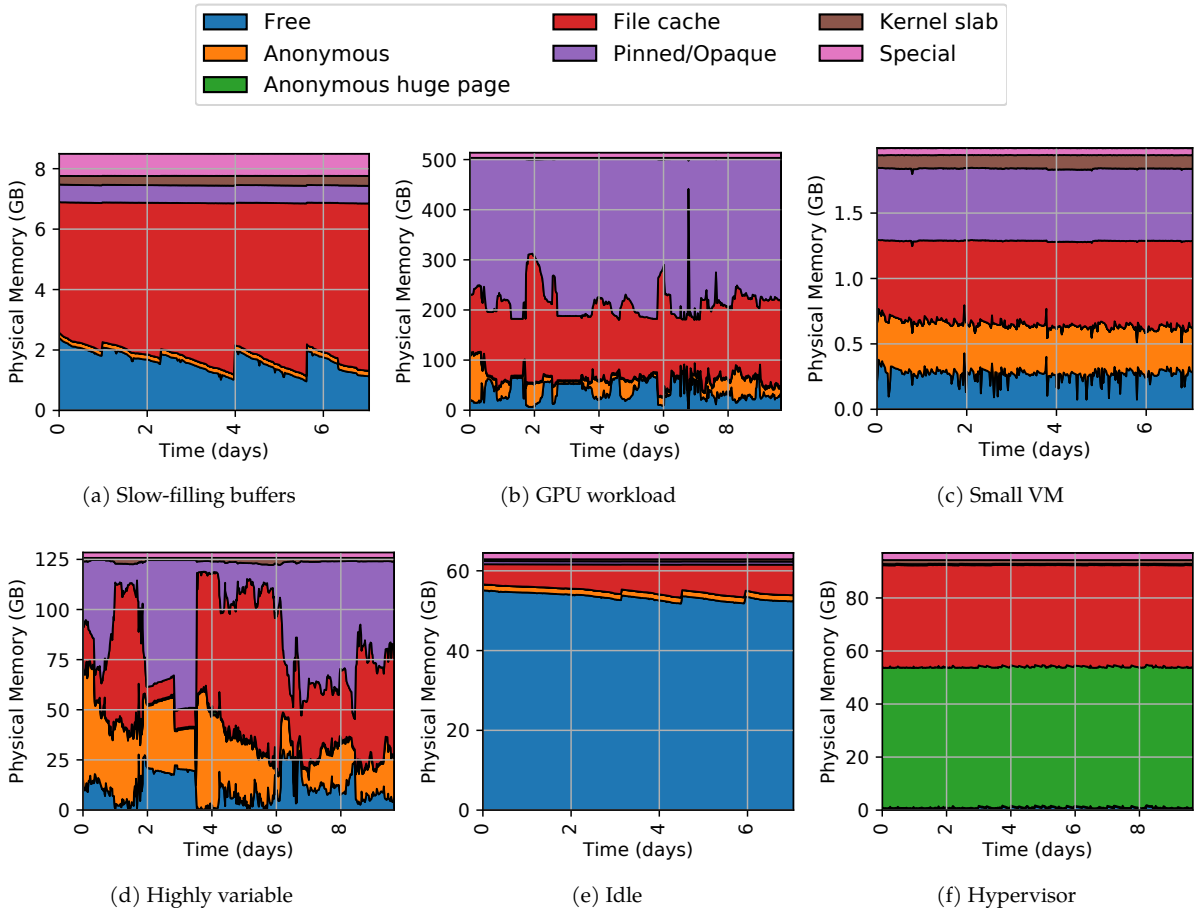
Memory Usage and Homogeneity

We characterize machines' use of physical memory, and analyze the contiguity of physical memory with the same page flags in `/proc/kpageflags` – which we call "homogeneous". Homogeneity is a good proxy for contiguity/fragmentation because contiguous memory allocations must be homogeneous in usage and page table permissions; non-homogeneous regions indicate discontiguous/fragmented memory usage, so homogeneity bounds contiguity.

Finding 3. *We observed 6 common and distinct memory usage patterns* (*Figure 5.1*).

In Linux, we found that all memory in the system can be classified as one of seven usage types: free/unallocated memory, anonymous (non-huge-page) memory, anonymous huge page memory, file cache memory, pinned/opaque memory (e.g., used by drivers/DMA), kernel slab memory (used for common kernel data structures), and other special cases (zero pages, pages with detected hardware failures, etc).

We observed 6 common memory usage patterns:



(a) Slow-filling buffers (b) GPU workload (c) Small VM

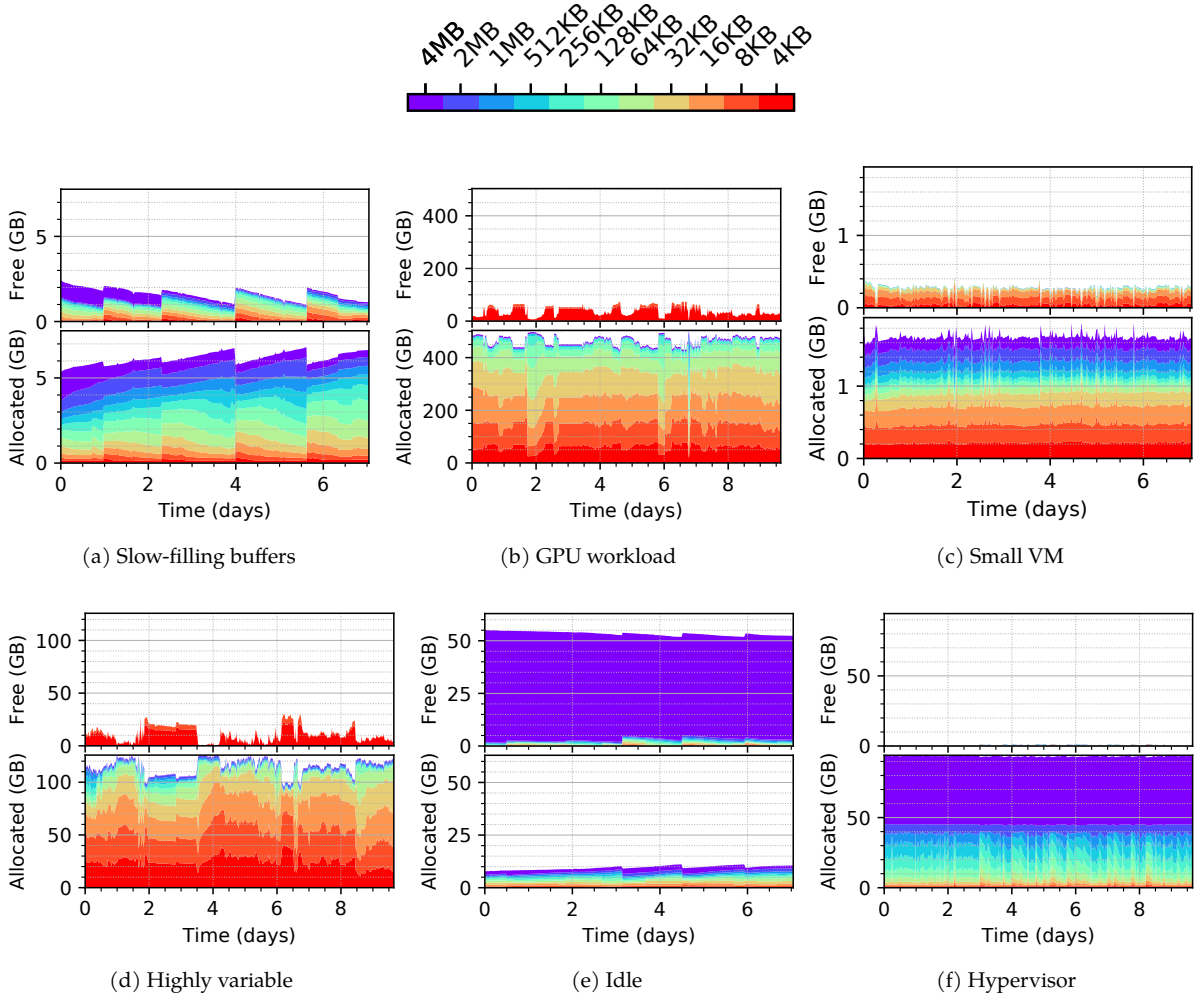
(d) Highly variable (e) Idle (f) Hypervisor

Figure 5.1: Examples of memory usage patterns.

1. *Slow-filling buffers*: slow-filling buffers that periodically flush to disk.
2. *GPU workloads*: dozens of gigabytes of memory dedicated to I/O.
3. *Small VMs with low-medium load*: small single-task dedicated VMs with fairly consistent memory usage, only a few hundred MBs of free memory, and significant memory dedicated to balloon drivers [138].
4. *Highly variable and unpredictable workloads*: widely varying and unpredictable memory usage as compute tasks run, terminate, and are replaced.
5. *Idle*: large amounts of idle free memory; low activity.
6. *Hypervisors*: mostly huge pages backing guest memory.

Implication: Memory usage is highly diverse but often predictable. While current designs assume workloads are unknown, often this assumption is overly conservative.

Finding 4. *Memory reclamation, triggered by memory pressure, is a major source of fragmentation because it breaks up homogeneous regions. The fragmentation is often irreparable, even when 1/4 of memory is freed periodically.*



(a) Slow-filling buffers (b) GPU workload (c) Small VM

(d) Highly variable (e) Idle (f) Hypervisor

Figure 5.2: Amount of contiguity in free and allocated memory for the different memory usage patterns in Figure 5.1.

This is a key finding of our work. Memory pressure causes fragmentation to irreparably worsen by up to an order of magnitude, even if up to 1/4 of total memory is periodically freed, supporting prior findings [96]. Often such freed pages are quickly allocated to the file cache, preventing them from being merged. Batch tasks and background daemons/loggers can cause regular and periodic memory pressure, leading to increasing fragmentation over time. These behaviors are especially clear in patterns (1) - (4), as exemplified in Figures 5.2a to 5.2d. Unfortunately, as shown in Section 5.2, memory pressure is common.

Conversely, homogeneity (of both allocated and free memory) is correlated to free memory ($R^2 = 0.63$, i.e., free memory is the single most important factor in amount of homogeneity). Machines that are never fully utilized tend to retain significant amounts of homogeneity in both free and allocated memory. Idle machines tend not become significantly more or less fragmented over time; there is no memory pressure to

trigger Linux's reclamation (increasing fragmentation) or compaction (decreasing fragmentation). Also, most idle machines lack significant fragmentation in their free memory anyway.

Implication: Reclamation must be made contiguity-aware, so that it does not needlessly break up memory regions.

Finding 5. *File caches comprise significant memory usage on most machines. Memory allocated to file caches usually becomes progressively more fragmented over time, especially in the presence of reclamation.*

This is a key finding of our work, illustrated by corresponding Figures 5.1 and 5.2. In patterns (1), (2), and (4), significant file cache activity leads to reclamation, which as we show later, increases fragmentation (illustrated by corresponding Figure 5.2). Moreover, Linux lacks general huge page support in the file cache; the file cache manages memory at base page granularity, so memory management operations often obliviously break up "incidentally contiguous" [118] memory regions. All patterns except (5) use significant amounts of file cache memory, so file cache memory has a significant role in fragmentation control.

Implications: Kernels should make file caches contiguity-aware. File cache huge page support would reduce fragmentation and allow this significant memory pool to use huge pages. Interestingly, most prior work focuses on anonymous huge pages [67, 87, 105, 113, 115, 142, 144]. Additionally, pages that reduce fragmentation can be prioritized for eviction from the file cache, including during writeback operations. For example, the kernel can flush dirty pages that interrupt a run of otherwise clean pages, or reclaim pages that are already not physically contiguous with their neighbors so as to avoid worsening fragmentation.

Finding 6. *Kernel slab memory and page tables usually accounted for very little memory usage.*

Combined, slab and page table memory comprised $\leqslant$ 5% on 44% of machines and $\leqslant$ 13% on all but 6 machines.

Implication: While some prior work significantly modifies the memory allocator to avoid fragmentation by these pages [115], it may suffice (and be more efficient) to reserve a small dedicated memory pool for slabs and page tables, falling back to the normal allocator in case of higher usage.

Finding 7. *For each finding above, there was one or two exceptional nodes that broke the rule.*

For example, there was one node that had high utilization and but also relatively high homogeneity. There was also a small but significant minority of nodes that did not fit any of the six patterns we identified, and a handful of nodes that had sizable amount of kernel slab memory.

Implications: It may make sense to have specialized memory management policies for common behaviors, but general fallback policies are needed for exceptions.

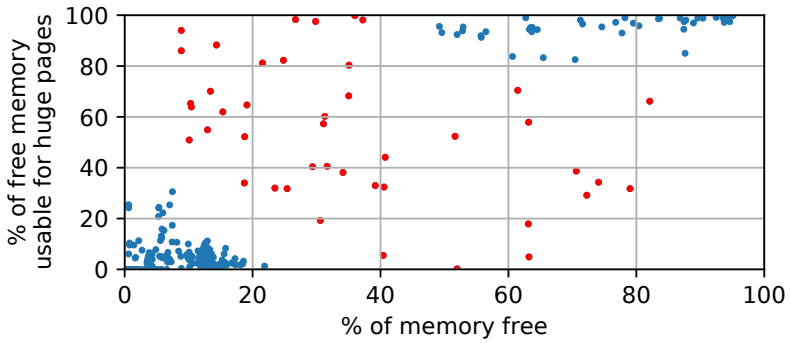


Figure 5.3: Median % of free memory vs median % of free memory that is contiguous enough for a 2MB huge page throughout the observation window for each machine. The 205 blue points indicate either low-memory-low-huge-page nodes or low-usage-high-huge-page nodes. The 43 red points indicate all other nodes.

Huge Page Feasibility and Usage

We examine huge page usage and the percentage of homogeneous regions that are 2MB or larger, a common huge page size on x86 and ARM.

Finding 8. *Few machines used huge pages.*

Only 69 machines used more than 10MB of huge pages at any time throughout the observation window, and many of those machines did not use huge pages most of the time. Only 35 machines used more than 1GB of huge pages at any time throughout the observation window. Applications do not usually request huge pages explicitly, and most machines are configured not to use huge pages, surprisingly. Except for hypervisor nodes, no machine had more than 1/4 of their memory backed by huge pages (and often much less). Hypervisor nodes used huge pages for over 1/2 of memory, presumably to back guest pages.

Implication: Kernels need transparent huge page systems that can be on-by-default without regressing performance.

Finding 9. *Large-scale contiguity (e.g., huge pages) is rare, but smaller-scale contiguity is quite common.*

Figure 5.2 exemplifies this finding. Small VMs usually have homogeneity up to 64KB, but rarely large than that. High-variability machines, such as HPC compute nodes (e.g., Figure 5.2d), rarely had homogeneity greater than 128KB, especially in free memory. Only when a task consuming more than 1/3 of memory terminated did fragmentation decrease, and then only 1/6 of memory became less fragmented. Moreover, the unfragmented memory was often quickly consumed by Linux's Transparent Huge Page system, leaving the remaining free memory fragmented. Kernel slab memory tends to have 8KB-16KB contiguity generally, which matches our findings in the previous section.

Implications: Our findings support prior proposals for contiguity-aware TLBs [116,118]. They also suggest

a 64KB granularity for some kernel mechanisms, as such homogeneous chunks are commonly present. Such mechanisms might include memory ballooning/hot-unplugging, reclamation, range-based virtual-address translations or permissions, or even the kernel allocator. Moreover, a smaller intermediate 64KB huge page size might be more usable than existing sizes (2MB, 1GB on x86_64), which often require compaction.

Finding 10. *On machines with little free memory, little of that free memory was usable for huge pages.*

Figure 5.3 shows the amount of free memory on each machine that is contiguous enough to create a huge page compared to the total amount of free memory. Over 83% of machines fit in one of two general patterns: (1) if $\leqslant$ 25% of memory is free, most of it is discontiguous (lower left); or (2) if $\geqslant$ 50% of memory is free, almost all of it is highly contiguous (upper right).

43 nodes (17%; red in the figure) fit neither pattern. Of these, most had a relatively large amount of file cache memory at some point. Many were also machines with highly variable workloads. A few also had relatively large amounts of slab memory (shown to cause fragmentation in the past [115]) and huge pages (i.e., contiguous memory was already being used, leaving free memory fragmented).

The use of file cache pages on an otherwise idle machine can reduce contiguity, as discussed in Finding 5. For example, about 15% of large machines with ample free memory had low contiguity compared to similar machines.

Implication: Currently in Linux, to have contiguity a machine must be either overprovisioned with memory or pay the overhead of memory compaction.

Change over time

We examine the usage of each page of memory on each machine across snapshots.

Finding 11. *The page flags of $\geqslant$ 95% of pages changed less than once every 3 hours, and on most machines, 20-60% of pages did not change at all.*

On almost all machines, $\geqslant$ 95% of pages changed usage (i.e., flags in `kpageflags`) $\leqslant$ 50 times over the course of a week (i.e., less than 1 change every 3 hours). Moreover, on most machines, 20-60% of pages did not change usage at all during the observation period. While it is possible that additional changes were not captured between snapshots, we took our snapshots at a random time in each half hour window so as not to miss frequent or periodic behavior.

For pages that did change, the changes tended to be slightly bursty, rather than uniform over time – on most machines, the distribution of time between changes was long-tailed (skewness between 2 and 4[1]), indicating multiple relatively rapid changes followed by extended periods with no change. Only 19 machines

[1] *Skewness* is a statistical quantification of the long-tailed-ness of a distribution. 0 indicates a symmetric distribution, while positive and negative values indicate right- or left-tailed-ness, respectively. A common rule of thumb is that values $\geqslant 1$ or $\leqslant -1$ indicate highly skewed distributions [34].

had a high rate of page changes. These machines had little free memory and bursty workloads that drove significant file cache activity.

Implications: This finding suggests that fragmentation is disproportionately affected by a small set of rapidly-changing pages. If so, then fragmentation control can be more efficient by focusing on these pages, rather than scanning through all pages, as current systems do. However, further investigation is needed; because we captured snapshots at an average interval of 30 minutes, it is possible that our methodology misses important changes in between snapshots that affect fragmentation's evolution over time without changing overall page usage (e.g., the flushing of dirty pages in the file cache).

Uptime

Finding 12. *Uptime has little to do with fragmentation; rather memory pressure is a much more important factor than uptime in magnitude of memory fragmentation.*

For security reasons, over half of DI machines restart weekly, while 91% were restarted within two weeks. In contrast, HPC machines had long and highly variable uptimes; over 97% of machines had an uptime over one month, and nearly 20% of machines had an uptime between 1 and 4 years. Many DI machines have low contiguity despite short uptimes due to their small size and memory pressure. Moreover, many of them had been rebooted just days before our observation window but were already as fragmented as long-uptime machines, despite very modest load and often entirely due to background processes such as logging or virus scans. In contrast, many non-idle HPC machines had significantly longer uptimes but comparable or better memory contiguity.

Implications: While some prior work suggests rebooting as a fragmentation control measure [17], our results show that this technique is unreliable at best.

5.3 Future Work

There are many aspects of our data that we have yet to explore. First, we do not analyze the kernel's "dirty" page flag (not to be confused with the page table bit), which indicates that a page needs to be flushed. Likewise, the kernel has an "accessed" flag (again not to be confused with the page table bit), which indicates that a page has been touched recently. While these flags give low-resolution information, they may still offer significant insights about idle memory and file cache dirtiness. Key questions include (1) are there common patterns of file cache usage? (2) what percentage of memory is idle? (3) are there policies that could be applied for different usage patterns? (4) what is needed to detect these patterns and apply appropriate

kernel policies? (5) what is the rate of change between snapshots (i.e., is there behavior between snapshots that our data does not capture)?

Second, our analysis of change over time is only cursory. Key questions include (1) can we identify changes in workload behavior over time? (2) how long does it take system behavior to quiesce to a steady state? (3) how does change over time correlate to other memory usage characteristics we analyzed?

Another avenue of future work is to prototype and test designs/policies suggested by our findings. For example, our work suggests that managing memory at a granularity around 16KB may be possible because small-scale contiguity is often available even on fragmented systems. Also, making reclamation and file cache management more contiguity-aware may reduce fragmentation. More research is needed to see if these ideas can work and benefit systems in practice.

5.4 Conclusion

More work is need to understand and mitigate fragmentation in current systems. Our work suggests that huge page support in the file cache and fragmentation-aware reclamation algorithms could have a big impact to reduce fragmentation.

6 EXPERIENCES AND TAKEAWAYS

In this section, we describe experiences, impressions, and takeaways that developed over the course of the work described in this dissertation.

Systems have a "zone of stability" corresponding to their common case usage. Outside the "zone of stability", they are unpredictable and unreliable. The F-16 fighter jet is "aerodynamically unstable" – it requires the fly-by-wire system to maintain an angle-of-attack within a certain window; outside of the window, the plane stalls. In our experience, modern systems are much the same: they are heavily tested and optimized along commonly used code paths, but if you stray off the beaten path and do something unusual, you are in for a crash (pun intended). Nowhere was this more evident than in our work on `0sim`. We found that while Linux has no theoretical or fundamental limits on the amount of physical memory, there are a lot of practical issues and annoyances that need sorting out before a researcher can be productive.

First, many Linux distributions and tools have arbitrary hard-coded limits. For example, Ubuntu does not support more than 1023GB of memory, and CentOS does not support more than 12TB. While CentOS documents its limit, Ubuntu, Debian, and Fedora do not document their limits, and we discovered them after long searches for an explanation for strange errors and crashes. QEMU's command line argument parser seems unable to accept "8000G" as an argument for the `-m` flag, which specifies the amount of memory of the guest, although it accepts smaller amounts. This artificially limits the size of `0sim` simulations to 8TB.

Second, documentation from hypervisors and chip vendors is sparse, especially regarding extreme environments like those we studied. We find that neither Intel nor AMD document the number of physical and virtual address bits their processor support; rather, we rely on Linux's `/proc/cpuinfo` file to find these values for machines to which we have access. Likewise, we found that knowledge about the meanings and usages of various KVM options was scarce. Often, we resorted to reading the KVM source code or crawling through poorly-organized kernel mailing lists. It is especially difficult to find out which KVM options are affected by the number of physical or virtual address bits supported by the host processor.

Third, problems rarely result in a clean error. For example, we found that if a guest was given 1TB or more on an Ubuntu host, no error was thrown at all; rather, the QEMU process itself hangs forever. In other situations, we found that booting a large guest on a CentOS host with the wrong KVM options would lead to a guest general protection fault on boot. In some situations, a seemingly sane combination of host, guest, and KVM options would inexplicably lead to guest kernel null pointer panics. These sorts of errors are hard to explain and generally difficult to debug.

We hope that by allowing researchers to play around with large simulated systems, `0sim` will improve

the quality of software, documentation, and failure behavior in general.

Fail-slow failures are growing in importance and should be treated as first-class correctness problems. Performance debugging remains among the most tedious and challenging of tasks facing system developers. It is not a stretch to say I spent most of the last few years debugging anomalous performance, often fruitlessly. In fact, anomalous performance during the 0sim project was the inspiration for the CBMM project.

I posit that the performance debugging curse occurs because current systems treat optimizations as implementation details to be hidden, rather than part of the interface of a system to be tested and documented. As a result, when the conditions of those hidden optimization are not met, the system is mysteriously slow, and the problem is difficult to debug, reproduce, or fix. An example of this is Transparent Huge Pages, as discussed in Chapter 4.

The problem is not limited to MM, either; I think it is a problem of growing importance to system design in general. In 2018, the architecture and OS communities were rocked by the Spectre and Meltdown vulnerabilities [86, 93]. These vulnerabilities exist solely because performance is not part of the architecture (i.e., interface) but the micro-architecture (i.e., implementation). Likewise, recent work on disk failures suggests that fail-slow disk failures have a massive impact on datacenters [94]. So far, system developers have treated optimizations as secondary-concerns compared to functional correctness. I suggest that we start treating system performance as part of the interface – a first-class correctness concern.

More practically, here are some properties I found myself repeatedly wishing were true of Linux:

- Transparency: optimizations should be strictly opt-in with good documentation about when they can be applied and what their limitations are. I include micro-architectural optimizations in this.
- Fail-fast: a system should prefer to fail quickly with an error than to succeed slowly without an error. Fast failures are easy to debug and reproduce. Slow successes propagate deeper into a system leading to either anomalous behavior (e.g., a really slow successful response) or delayed errors (e.g., a timeout in some other part of the system).
- Configurability: it should be possible to turn on and tune a single optimization in the absence of others to see what its effects are (and again, I include micro-architectural optimizations). For example, such configurability would have helped to debug the performance overhead of asynchronous memory compaction; instead, we resorted to coaxing the system into compaction, which also had the effect of triggering reclamation and confounding our experiments. Another example: many times, I've wanted to turn off hardware optimizations such as page walk caches or prefetchers for a short stint so as to see the true impact of TLBs or NUMA effects.

Heuristics are a fast way to make a project functional, but they can result in complications later when hidden assumptions interact pathologically. By their nature, heuristics abstract away details, simplifying implementation. However, those same details may turn out to matter, leading to unexpected behavior that is hard to debug precisely because the details are hidden. Worse, multiple heuristics together may produce pathological interactions, as we saw in Chapter 4.

A major thrust of our work focuses on measurement-based MM designs and policies. If you use heuristics, make sure you thoroughly measure, understand, and document your assumptions. As demonstrated by CBMM, this yields faster and more debuggable systems.

MM extensibility and modularity are important. Code quality and engineering effort matter a lot, even though they often go unrewarded. Linux's current MM code is quite initimidating, in my humble opinion, and I've been working with it for years. The code is mostly not modular, documented, or self-documenting. Pointers to every data structure are scattered around like dandelions. Complex and non-obvious synchronization invariants are enforced by an ominous mix of convention, grep, and luck (especially if you are writing a research prototype that will never be code-reviewed). `struct page`, one of the core data structures of Linux MM, has about 20 nested unions of structs of unions of madness – to the extent that key developer Matthew Wilcox made a spreadsheet to explain it [141]. This lack of modularity and extensibility is a strong hinderance to experimentation.

Kernel developers must wrestle with complex, invasive changes to kernel MM to support new (or even not-so-new) hardware paradigms. For example, even though huge pages have been generally available for over two decades [105], Linux still does not implement huge page support in its file cache. Efforts to add this support necessitate massive refactorings, such as the `struct folio` work currently under way [48]. Other recent work such as Meta's Transparent Page Placement have made invasive changes across 22 files to implement support for memory tiering [72, 99].

The lack of extensibility also makes it difficult to measure kernel behavior without modifying it directly to add in instrumentation. This was the approach we resorted to for our CBMM study of page fault latency and huge page benefits. Admittedly, the eBPF framework is massively and rapidly improving kernel extensibility, but there are many more niche tasks that cannot be done with generic kernel extentions like eBPF. For example, in our attempts to reproduce fragmentation patterns, the lack of MM extensibility proved to be a hinderance to studying and reproducing MM behavior.

Finally, unmodular and unextensible code is hard to understand and learn about, which particularly affects students. I spent a fair bit of time during the 0sim and CBMM projects helplessly floundering around the kernel MM code hoping for answers about why my kernel was panicking. I can recall watching at least 3

other students go through the same process. This is a massive and unnecessary waste of time, grant money, and sleepless nights.

I am heartened by the renewed emphasis that modern systems programming languages, such as modern C++ and Rust, place on modularity and encapsulation. I would like to see more of these concepts seep back into the Linux kernel. For my part, I have tried to spend extended time documenting and cleaning up the code in my paper artifacts and research prototypes in hopes that they will make life easier for the next student that has to work with them.

7 CONCLUSIONS AND FUTURE WORK

Memory management techniques are struggling to keep up with modern computing needs. Supporting new hardware and software paradigms mandates new tools and new MM designs and policies. In our work, we approached this mandate with three thrusts.

First, we found that many current algorithms are not scalable to large-memory systems. We showed that 0sim is able to help find and debug these issues and prototype solutions.

Second, we found that the intuition-based heuristics and "scattered" implementations used in many MM designs cause performance anomalies. In CBMM, we propose a centralized design based on thorough measurements of system behavior that produces more consistent performance and is easier to debug.

Third, we characterized physical memory fragmentation in the wild by observing live machines. We found a wide variety of behaviors and identified page reclamation and the file cache as key instigators of fragmentation in Linux.

More complex hardware architectures and larger-scale deployments will only make MM harder and more performance-critical. Our work, while preliminary, demonstrates a critical need for thorough measurement and principled MM designs based on concrete data, rather than intuition, heuristics, or implementation simplicity.

Our work suggests several useful directions for future exploration...

First, there is a need for MM algorithms and designs that scale with memory capacity. While 0sim identified several potential issues, finding solutions is difficult. Some prior work has suggested filesystem algorithms for memory management, as filesystems regularly have large capacities [130]. We found that a number of algorithms in current MM designs involve scanning through lists of pages, including reclamation and compaction. Prior work and our work suggest that these algorithms are already becoming too expensive; can they be replaced with filesystem-inspired algorithms? Or obviated entirely?

Second, there is a need to design software such that implicit assumptions and limitations become explicit with clear and actionable failure messages. We found that system software often breaks when run in uncommon settings or with uncommon inputs. For example, in our 0sim work, we experienced odd software failures from QEMU, KVM, and a number of Linux distributions when trying to run them with large amounts of memory. The failures ranged from (suspected) emulated hardware failures to command line argument parsing failures, and the mode of failure was usually user-unfriendly. What is needed to avoid such failure

modes? Better programming language design? Or better programming frameworks/paradigms in existing languages? Or something else?

Third, there is a need for extensible MM. We found that implementing new MM features in Linux for experimentation is difficult; it hinders exploration and wastes time. There is a need for extensible MM designs that allow the exploration of different policies and mechanisms. For example, during the CBMM project, we wished to replace current MM policies with cost-benefit-based policies, but the implementation of Linux MM makes this difficult because policies are scattered all over the code. Likewise, the lack of extensibility made it difficult to accurately reproduce fragmentation patterns without extensive kernel modifications. Is it possible to make different kernel MM mechanisms (e.g., the physical memory allocator) and policies (e.g., huge page usage or page placement) replacable with kernel modules or eBPF scripts or some other mechanism?

Fourth, there is a need for new and more flexible hardware performance introspection, both for debugging and for guiding MM policies. Page table accessed/dirty bits are woefully insufficient; MM systems need more and better and cheaper information to guide their actions. In our dream world, hardware counters would be easily configurable and filterable. For example, the kernel would be able to measure events that satisfy certain conditions, such as page walks or TLB misses related to in a particular region of memory (e.g., is this data structure causing TLB misses?). It would also be nice to have visibility into the current and near-future state of hardware... What translations are currently in the TLB? How long would a TLB miss on a particular address take? What are the N most (or least) heavily accessed pages? Is this page in the LLC cache? How much memory bandwidth is currently being used by communication between NUMA nodes? Currently, these questions are difficult or expensive to answer, but they are exactly the sorts of questions memory managers need to ask.

Fifth, there is a need for better huge page management and memory reclamation/migration algorithms. This has been an area of increasingly active research in recent years. Memory compaction, migration (e.g., between NUMA nodes or memory tiers), and reclamation have well-known overheads associated with them that make them undesirable. And yet these systems are increasingly critical to efficiency and good performance in modern systems. Our CBMM and fragmentation studies indicate that future MM requires robust and low-overhead anti-fragmentation techniques to make huge pages usable. Likewise, memory reclamation, and memory migration generally, require low-overhead ways to identify target pages.

Moreover, all of these algorithms require low-overhead ways to copy memory contents and move page mappings. Current designs require unmapping, modifying page tables, flushing TLBs (and possibly sending

IPIs to do so), remapping, and taking TLB misses. Perhaps a hardware extension can provide a low-overhead way to change a page mapping?

Sixth, there is an opportunity to use the relative regularity of first-party datacenter workloads for fun and profit. First-party datacenter workloads commonly run with high redundancy for long amounts of time or at high frequencies. They are well-known to the operator, highly controlled, and change at known times. And yet kernels treat all workloads as if they are the first to run the workload ever in the history of mankind. It seems that there is an opportunity for automated profiling or policy exploration at the cluster level.

In particular, we believe that cluster-level A/B testing can be used to test out different MM policies. For example, the cluster manager can assign some machines to use huge pages for a particular application or memory region, while other machines use base pages. The resulting performance difference can be used to assess whether huge pages help for the given workload or not. Similar questions about page placement policies or performance interference between applications can also be answered by this sort of cluster-level A/B testing.

The idea is not without precedent. Meta has used a similar automated A/B-style testing to fine-tune architectural and boot-time parameters [129]. Also, Google uses cluster-level data aggregation and machine learning to guide its software-defined far-memory system [88]. We propose doing something similar to guide MM policy cluster-wide.

Seventh, there is still ample data to mine from the fragmentation study. The data we collected for our fragmentation study is large, high-dimensional, and rich. While we extracted a number of findings from it, there is still much left to learn. For example, we did not analyze the effect of page dirtiness on file cache behavior. Or which pages are most likely to change over time. Or which workloads have the most volatile pages. Or how many of the changes we observed over time were suboptimal in retrospect. Each of these questions has potential to significantly improve memory management.

Our data and artifacts are publicly available and we hope they enable others to bring new insights to memory management.

BIBLIOGRAPHY

[1] SPEC CPU 2017 Benchmark Suite. https://www.spec.org/cpu2017/.

[2] Linux Kernel Documentation: Device Mapper Thin-Provisioning. https://www.kernel.org/doc/Documentation/device-mapper/thin-provisioning.txt.

[3] Linux Kernel Documentation: Kernel Samepage Merging. https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html.

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

[5] Redis latency problems troubleshooting. https://redis.io/topics/latency.

[6] Remove PG_ZERO and zeroidle (page-zeroing) entirely. https://news.ycombinator.com/item?id=12227874, August 2016.

[7] Bulent Abali, Hubertus Franke, Dan E. Poff, Robert A. Saccone, Charles O. Schulz, Lorraine M. Herger, and T. Basil Smith. Memory Expansion Technology (MXT): Software support and performance. *IBM Journal of Research and Development*, 45(2), March 2001.

[8] Nitin Agrawal, Leo Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Emulating Goliath Storage Systems with David. *ACM Trans. Storage*, 7(4), February 2012.

[9] Martin Aigner, Christoph M. Kirsch, Michael Lippautz, and Ana Sokolova. Fast, Multicore-scalable, Low-fragmentation Memory Allocation Through Large Virtual Memory and Global Data Structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2015.

[10] Alaa R. Alameldeen, Milo M. K. Martin, Carl J. Mauer, Kevin E. Moore, Min Xu, Mark D. Hill, David A. Wood, and Daniel J. Sorin. Simulating a $2M Commercial Server on a $2K PC. *Computer*, 36(2), February 2003.

[11] alvinashcraft, v-kents, DCtheGeek, drewbatgit, and msatranjr. Large-Page Support. https://learn.microsoft.com/en-us/windows/win32/memory/large-page-support, January 2021.

[12] Amazon Inc. EC2 High Memory instances with 18TiB and 24TiB of memory are now available with On-Demand and Savings Plan purchase options. https://aws.amazon.com/about-aws/whats-new/

2022/10/ec2-high-memory-instances-18tib-24tib-memory-available-on-demand-savings-plan-purchase-options/.

[13] Amazon Inc. EC2 Instance Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[14] Amazon Inc. Amazon EC2 High Memory Instances with 6, 9, and 12 TB of Memory, Perfect for SAP HANA. https://aws.amazon.com/blogs/aws/now-available-amazon-ec2-high-memory-instances-with-6-9-and-12-tb-of-memory-perfect-for-sap-hana/, September 2018.

[15] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks. Architecture of the IBM system/360. *IBM Journal of Research and Development*, 8(2), April 1964.

[16] Apple Inc. OS X Mavericks Core Technologies Overview. https://images.apple.com/media/us/osx/2013/docs/OSX_Mavericks_Core_Technology_Overview.pdf, 2013.

[17] Jean Araujo, Rubens Matos, Paulo Maciel, Rivalino Matias, and Ibrahim Beicker. Experimental evaluation of software aging effects on the eucalyptus cloud computing infrastructure. In *Proceedings of the Middleware 2011 Industry Track Workshop*, Middleware '11, December 2011.

[18] Vlastimil Babka. Overview of memory reclaim in the current upstream kernel. In *Linux Plumbers Conference 2021*, September 2021.

[19] David H Bailey, E. Barszcz, John T Barton, D. S. Browning, R. L. Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Tom A Lasinski, Robert S Schreiber, Horst D Simon, V. Venkatakrishnan, and Sisira K Weeratunga. The NAS Parallel Benchmarks: Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, 1991.

[20] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. 2013.

[21] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA, 2013.

[22] Leland L. Beck. A Dynamic Storage Allocation Technique Based on Memory Residence Time. *Commun. ACM*, 25(10), October 1982.

[23] C. Gordon Bell and Ike Nassi. Revisiting Scalable Coherent Shared Memory. *Computer*, 51(1), January 2018.

[24] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2010.

[25] Anna Bendersky and Erez Petrank. Space Overhead Bounds for Dynamic Memory Management with Partial Compaction. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2011.

[26] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Mor Harchol-Balter, and Siddhartha Sen. RobinHood: tail latency-aware caching – dynamic reallocating from cache-rich to cache-poor. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, October 2018.

[27] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *SIGPLAN Not.*, 35(11), November 2000.

[28] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 1st edition, 2016.

[29] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating Two-dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2008.

[30] Christian Bienia. *Benchmarking modern multiprocessors*. PhD thesis, Princeton University, January 2011.

[31] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2), May 2011.

[32] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an Experiment in Operating System Structure and State Management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[33] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Trans. Comput. Syst.*, 15(4), November 1997.

[34] M.G. Bulmer. *Principles of Statistics*. Dover Books on Mathematics Series. Dover Publications, 1979.

[35] Shakeel Butt, Suren Baghdasaryan, and Yu Zhao. Finding more DRAM, September 2019.

[36] Richard W. Carr and John L. Hennessy. WSCLOCK – a simple and effective algorithm for virtual memory management. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, December 1981.

[37] Matthew Chapman and Gernot Heiser. vNUMA: A Virtual Shared-memory Multiprocessor. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX ATC, 2009.

[38] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys, 2013.

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

[40] Jonathan Corbet. Persistent memory and page structures. `https://lwn.net/Articles/644079/`, May 2015.

[41] Jonathan Corbet. Persistent memory support progress. `https://lwn.net/Articles/640113/`, April 2015.

[42] Jonathan Corbet. ZONE_DEVICE and the future of struct page. `https://lwn.net/Articles/717555/`, March 2017.

[43] Jonathan Corbet. Improving support for large, contiguous allocations. `https://lwn.net/Articles/753167/`, May 2018.

[44] Jonathan Corbet. Ktask: Optimizing CPU-intensive kernel work. `https://lwn.net/Articles/771169/`, November 2018.

[45] Jonathan Corbet. Toward better performance on large-memory systems. `https://lwn.net/Articles/753171/`, May 2018.

[46] Jonathan Corbet. Proactively reclaiming idle memory. `https://lwn.net/Articles/787611/`, May 2019.

[47] Jonathan Corbet. Multi-generational LRU: The next generation. `https://lwn.net/Articles/856931/`, May 2021.

[48] Jonathan Corbet. A memory-folio update. `https://lwn.net/Articles/893512/`, May 2022.

[49] Couchbase. Disabling Transparent Huge Pages (THP) | Couchbase Docs. https://docs.couchbase.com/server/current/install/thp-disable.html.

[50] Jean-Francois Dagenais. Extra large DMA buffer for PCI-E device under UIO. https://lkml.org/lkml/2011/11/18/462.

[51] Dan Williams. Randomize free memory. https://lwn.net/Articles/767614/.

[52] D. Julia. M. Davies. Memory Occupancy Patterns in Garbage Collection Systems. *Commun. ACM*, 27(8), August 1984.

[53] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2), February 2013.

[54] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the Sixth Conference on Symposium on Operating Systems Design & Implementation*, OSDI, 2004.

[55] Srividya Desireddy. [PATCH v2] zswap: Zero-filled pages handling. https://lkml.org/lkml/2017/8/16/560.

[56] Cort Dougan, Paul Mackerras, and Victor Yodaiken. Optimizing the idle task and other MMU tricks. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, February 1999.

[57] Lars Eggert, Alan Cox, Cort Dougan, and Matt Dillon. clearing pages in the idle loop, July 2000.

[58] Magnus Ekman and Per Stenstrom. A Robust Main-Memory Compression Scheme. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA, 2005.

[59] Jakob Engblom. Simulating six terabytes of serious RAM. https://software.intel.com/en-us/blogs/2016/09/02/simulating-six-terabytes-of-serious-ram, 2017.

[60] Jason Evans. Scalable memory allocation using jemalloc. https://engineering.fb.com/2011/01/03/core-data/scalable-memory-allocation-using-jemalloc/, January 2011.

[61] Michael Joseph Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, Henry M Levy, and Chandramohan A Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP, 1995.

[62] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. BadgerTrap: a tool to instrument x86-64 TLB misses. *ACM SIGARCH Computer Architecture News*, 42(2), September 2014.

[63] Jayneel Gandhi, Vasileios Karakostas, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Ünsal. Range Translations for Fast Virtual Memory. *IEEE Micro*, 36(3), May 2016.

[64] Erol Gelenbe, J. C. A. Boekhorst, and J. L. W. Kessels. Minimizing Wasted Space in Partitioned Segmentation. *Commun. ACM*, 16(6), June 1973.

[65] Google Inc. Borg cluster traces from Google. `https://github.com/google/cluster-data`.

[66] Google Inc. Google Compute Engine Pricing - Google Cloud. `https://cloud.google.com/compute/pricing#machinetype`.

[67] Mel Gorman and Patrick Healy. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, June 2008.

[68] Mel Gorman and Andy Whitcroft. The what, the why and the where to of anti-fragmentation. In *Proceedings of the Linux Symposium*, volume 1, January 2006.

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

[70] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To Infinity and Beyond: Time Warped Network Emulation. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP Poster Session, 2005.

[71] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, October 2017.

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

[73] Daniel S. Hirschberg. A Class of Dynamic Memory Allocation Algorithms. *Commun. ACM*, 16(10), October 1973.

[74] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and M. West. Scale and performance in a distributed file system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, November 1987.

[75] A. H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *Proceedings of the 17th USENIX Conference on Operating Systems Design and Implementation*, OSDI '21, 2021.

[76] Intel Inc. 5-Level Paging and 5-Level EPT. `https://software.intel.com/en-us/download/5-level-paging-and-5-level-ept-white-paper`.

[77] Intel Inc. Timestamp-Counter Scaling (TSC scaling) for Virtualization. `https://www.intel.com/content/www/us/en/processors/timestamp-counter-scaling-virtualization-white-paper.html`.

[78] Intel Inc. Intel's 3D XPoint™ Technology Products – What's Available and What's Coming Soon. `https://software.intel.com/en-us/articles/3d-xpoint-technology-products`, October 2017.

[79] John Wilkes. More Google Cluster Data.

[80] Mark S. Johnstone and Paul R. Wilson. The Memory Fragmentation Problem: Solved? In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, 1998.

[81] Daniel Jordan. [RFC,v4,00/13] ktask: Multithread CPU-intensive kernel work - Patchwork. `https://patchwork.kernel.org/cover/10668661/`.

[82] Frans Kaashoek, Robert Morris, and Yandong Mao. Optimizing MapReduce for Multicore Architectures. Technical Report MIT-CSAIL-TR-2010-020, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 2010.

[83] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, June 2015.

[84] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. Firesim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA, 2018.

[85] Kenneth C. Knowlton. A Fast Storage Allocator. *Commun. ACM*, 8(10), October 1965.

[86] Paul Kocher, Jann Horn, Anders Fogh, and Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy*, S&P '19, 2019.

[87] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, November 2016.

[88] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, April 2019.

[89] Christopher Lameter. Increase page fault rate by prezeroing V1 [0/3]: Overview. `https://lkml.org/lkml/2004/12/21/142`, December 2004.

[90] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015.

[91] Ted G. Lewis, Brian J. Smith, and Marilyn Z. Smith. Dynamic Memory Allocation Systems for Minimizing Internal Fragmentation. In *Proceedings of the 1974 Annual ACM Conference - Volume 2*, 1974.

[92] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'23, 2023.

[93] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium*, Security '18, 2018.

[94] Ruiming Lu, Erci Xu, Yiming Zhang, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Minglu Li, and Jiesheng Wu. NVMe SSD Failures in the Field: the Fail-Stop and the Fail-Slow. In *2022 USENIX Annual Technical Conference*, ATC'22, 2022.

[95] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based Memory Allocation for C++ Server Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, March 2020.

[96] Mark Mansi and Michael M. Swift. 0sim: Preparing System Software for a World with Terabyte-scale Memories. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, March 2020.

[97] Mark Mansi and Michael M. Swift. Policy/mechanism separation in the Warehouse-Scale OS, March 2023.

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

[99] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent Page Placement for CXL-Enabled Tiered Memory, June 2022.

[100] Microsoft Inc. Microsoft Azure Traces. `https://github.com/Azure/AzurePublicDataset`.

[101] Microsoft Inc. Pricing - Linux Virtual Machines | Microsoft Azure. `https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/`.

[102] MongoDB Inc. Disable Transparent Huge Pages (THP) — MongoDB Manual. `https://docs.mongodb.com/manual/tutorial/transparent-huge-pages`.

[103] Andrew Morton. Re: [PATCH - RFC] allow setting vm_dirty below 1% for large memory machines. `https://lkml.org/lkml/2007/1/9/80`.

[104] Andrew Morton. Re: [PATCH v2] z3fold: The 3-fold allocator for compressed pages. `https://lkml.org/lkml/2016/4/21/799`.

[105] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02, December 2002.

[106] Michal Nazarewicz. A deep dive into CMA. `https://lwn.net/Articles/486301/`.

[107] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, 6(2), March 2008.

[108] Norman R. Nielsen. Dynamic Memory Allocation in Computer Simulation. *Commun. ACM*, 20(11), November 1977.

[109] Markus F.X.J. Oberhumer. Oberhumer.com: LZO real-time data compression library. `http://www.oberhumer.com/opensource/lzo/`.

[110] OpenAFS Foundation. OpenAFS. `https://www.openafs.org/`.

[111] Oracle Inc. Database Installation Guide. `https://docs.oracle.com/cd/E11882_01/install.112/e47689/pre_install.htm#LADBI1152`.

[112] Oracle Inc. HotSpot Virtual Machine Garbage Collection Tuning Guide. `https://docs.oracle.com/en/java/javase/11/gctuning/z-garbage-collector1.html#GUID-A5A42691-095E-47BA-B6DC-FB4E5FAA43D0`.

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

[114] Ashish Panwar, Naman Patel, and K. Gopinath. A Case for Protecting Huge Pages from the Kernel. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys, 2016.

[115] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, 2018.

[116] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. Hybrid TLB Coalescing: Improving TLB Translation Coverage Under Diverse Fragmented Memory Allocations. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, 2017.

[117] James L. Peterson and Theodore A. Norman. Buddy Systems. *Commun. ACM*, 20(6), June 1977.

[118] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced Large-Reach TLBs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, December 2012.

[119] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized environments: can you have it both ways? In *Proceedings of the Forty-Eighth International Symposium on Microarchitecture*, MICRO-48, 2015.

[120] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. Mesh: compacting memory management for C/C++ applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '19, June 2019.

[121] Aravinda Prasad and K. Gopinath. Prudent Memory Reclamation in Procrastination-Based Synchronization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2016.

[122] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *;login:*, 39(6), December 2014.

[123] Dennis M. Ritchie and Ken Thompson. The UNIX Time-sharing System. *Commun. ACM*, 17(7), July 1974.

[124] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20(3), January 1977.

[125] Karl Rupp. 40 Years of Microprocessor Trend Data. `https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/`.

[126] Marta Rybczyńska. Introducing maple trees. `https://lwn.net/Articles/845507/`.

[127] ScaleMP Inc. ScaleMP - Virtualization for high-end computing. `https://www.scalemp.com/`.

[128] John E. Shore. On the External Storage Fragmentation Produced by First-fit and Best-fit Allocation Strategies. *Commun. ACM*, 18(8), August 1975.

[129] A. Sriraman, A. Dhanotia, and T. F. Wenisch. SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, June 2019.

[130] Michael M. Swift. Towards O(1) Memory. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS, 2017.

[131] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. *ACM SIGPLAN Notices*, 29(11), November 1994.

[132] Huangshi Tian, Yunchuan Zheng, and Wei Wang. Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, November 2019.

[133] TidalScale Inc. Software Defined Servers. `https://www.tidalscale.com/technology`.

[134] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, April 2020.

[135] Linus Torvalds. Page zeroing strategy, December 2000.

[136] Linus Torvalds. Pre-populating anonymous pages. `https://www.realworldtech.com/forum/?threadid=185310&curpostid=185398`, June 2019.

[137] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Proceedings of the 16th Annual Middleware Conference*, Middleware, 2015.

[138] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02, 2002.

[139] Yang Wang, Manos Kapritsos, Lara Schmidt, Lorenzo Alvisi, and Mike Dahlin. Exalt: Empowering Researchers to Evaluate Large-scale Storage Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI, 2014.

[140] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, February 2022.

[141] Matthew Wilcox. [PATCH v4 00/16] Rearrange struct page. `https://lore.kernel.org/all/20180430202247.25220-1-willy@infradead.org/T/#u`.

[142] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation Ranger: Operating System Support for Contiguity-aware TLBs. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, 2019.

[143] Wenbo Zhang. Why We Disable Linux's THP Feature for Databases. `https://pingcap.com/blog/why-we-disable-linux-thp-feature-for-databases`, December 2020.

[144] Weixi Zhu, Alan L. Cox, and Scott Rixner. A Comprehensive Analysis of Superpage Management Mechanisms and Policies. In *2020 USENIX Annual Technical Conference*, ATC '20, 2020.