Operating System Support for Efficient Virtual Memory

A THESIS

SUBMITTED FOR THE DEGREE OF

Doctor of Philosophy

IN THE

Faculty of Engineering

BY

Ashish Panwar

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

February, 2022

Declaration of Originality

I, **Ashish Panwar**, with SR No. **04-04-00-10-12-18-1-15943** hereby declare that the material presented in the thesis titled

Operating System Support for Efficient Virtual Memory

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2018-2022**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Advisor Signature

DEDICATED TO

Dr. Akshi Vashistha and Aayansh Panwar
my wife and our son

Acknowledgements

I owe a lot of gratitude to many amazing people who have helped me directly or indirectly to come so far. This dissertation would be incomplete without acknowledging their support.

First and foremost, I want to thank my wonderful research supervisors Prof. K. Gopinath and Prof. Arkaprava Basu. They have taught me so many things over the years including how to conduct research and convey research ideas. Their depth and breadth of knowledge in computer science and otherwise amazes me to date. They have been the pillar of support during my tough times. Most importantly, I want to thank my advisors for showing immense faith in me and providing me the freedom to explore my ideas at different stages of this journey.

I want to extend many thanks to my co-authors whose contributions have been invaluable for this dissertation: Prof. Sorav Bansal, Prof. Abhishek Bhattacharjee, Prof. Timothy Roscoe, Dr. Jayneel Gandhi, Dr. Aravinda Prasad, Dr. Reto Achermann, Venkat Sri Sai Ram, Akash Panda, and Naman Patel. I have greatly benefited from our discussions over the years.

I want to thank VMware Research for having me as an intern during the summer of 2019. Special thanks to Dr. Jayneel Gandhi for being a wonderful support system and an incredible mentor during and after my internship. The three months spent at VMware was a wonderful learning experience for me where I got to interact with esteemed researchers from the computer systems community.

I thank all my friends from CSA and IISc, including friends from the Computer Architecture and Systems Lab (CASL) and Computer Systems Lab (CSL). This includes current and former students: Naman Patel, Sandeep Kumar, Priyanka Singla, Poorna, Aripth K, Ajinkya Rajput, Abhishek Dubey, Shweta Pandey, Ajay Ashok Nayak, Rajat Jain, Pratheek B, Neha Jawalkar, Sujay Yadalam, Jaya Jyothiswaroop Kotni, Ravi Shreyas Anupindi, Aditya K Kamath, Sandesh Singh Patel, and Kingshuk Majumdar. I will always cherish the moments we have spent together at IISc including our technical discussions as well as time spent in leisure activities such as playing in the gymkhana or having meals together. This journey has been memorable because of such wonderful friends.

I want to thank the staff members of the CSA department. Special thanks to Ms. Kushael,

Mrs. Padmavathi, Ms. Meenakshi, and Ms. Nishitha for their hard work and dedication that ensured the smooth handling of academic requirements. I thank Mr. Akhshay Nath for helping with technical requirements time and again.

I also want to thank the Confederation of Indian Industry, Government of India, and Microsoft Research Lab India for supporting me through the Prime Minister's Fellowship Scheme for Doctoral Research. I appreciate all the support received from my industry mentor Dr. Muthian Sivathanu from Microsoft Research. I thank Mr. Ravi Hira, Ms. Neha Gupta, and Ms. Shalini Sharma from CII for taking care of the formal requirements of the fellowship program.

Finally, I want to express my deepest gratitude to my entire family – my parents and sisters. I cannot thank my wife Dr. Akshi Vashistha enough. She has been the backbone of this thesis and I cannot imagine writing it without her endless love and support.

Abstract

Computers rely on the virtual memory abstraction to simplify programming, portability, physical memory management and ensure isolation among co-running applications. However, it creates a layer of indirection in the critical path of execution wherein the processor needs to translate an application-generated virtual address into the corresponding physical address before performing the computation. To accelerate the virtual-to-physical address translation, processors cache recently used addresses in Translation Lookaside Buffers (TLBs).

Unfortunately, modern data-centric applications executing on large memory servers experience frequent TLB misses. The processor services TLB misses by walking the in-memory page tables that often involves accessing physical memory. Consequently, the processor spends 30-50% of total cycles in servicing TLB misses alone for many big-data applications. Virtualization and non-uniform memory access (NUMA) architectures in multi-socket servers further exacerbate this overhead. Virtualization adds an additional level of address translation while NUMA can increase the latency of accessing page tables residing on a remote socket. The address translation overhead will increase further with deeper page tables and multi-tiered memory systems in newer and upcoming systems. In short, virtual memory is showing its age in the era of data-centric computing.

In this thesis, we propose ways to moderate the overhead of virtual-to-physical address translation. The majority of this thesis focuses on huge pages. Processor designers have invested significant hardware in supporting huge pages to reduce the number and cost of TLB misses e.g., x86 architecture supports 2MB and 1GB huge pages. However, we find that operating systems often fail to harness the full potential of huge pages. This thesis highlights the pitfalls associated with the current huge page management strategies and proposes various operating system enhancements to maximize the benefits of huge pages. We also address the effect of non-uniform memory accesses on address translation with NUMA-aware page table management.

A key objective of this thesis is to avoid modifying the applications or adding new features to the hardware. Therefore, all the solutions discussed in this thesis apply to current hardware and remain transparent to the applications. All of our contributions are open-sourced.

Publications based on this Thesis

1. **[ASPLOS'18] Making Huge Pages *Actually* Useful**
   Ashish Panwar, Aravinda Prasad and K. Gopinath
   In proceedings of the 23rd ACM International Conference on *Architectural Support for Programming Languages and Operating Systems*, Williamsburg, VA, USA, April, 2018.

2. **[ASPLOS'19] HawkEye: Efficient Fine-grained OS Support for Huge Pages**
   Ashish Panwar, Sorav Bansal and K. Gopinath
   In proceedings of the 24th ACM International Conference on *Architectural Support for Programming Languages and Operating Systems*, Providence, RI, USA, April, 2019.

3. **[MICRO'21] Trident: Harnessing Architectural Resources for All Page Sizes on x86 Systems**
   Ashish Panwar*, Venkat Sri Sai Ram* and Arkaprava Basu
   * Joint first authors.
   To appear in proceedings of the 54th IEEE/ACM *International Symposium on Microarchitecture*, October, 2021.

4. **[ASPLOS'21] Fast Local Page-Tables for Virtualized NUMA Servers**
   Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K. Gopinath and Jayneel Gandhi
   In proceedings of the 26th ACM International Conference on *Architectural Support for Programming Languages and Operating Systems*, April, 2021 (virtual).

Contents

Acknowledgements i

Abstract iii

Publications based on this Thesis iv

Contents v

List of Figures x

List of Tables xv

1 Introduction 1

1.1 Benefits of virtual memory . . . 1
1.2 The cost of virtual memory . . . 2
1.3 Problem statement . . . 5
1.4 Contributions of this dissertation . . . 6
1.5 Organization of this dissertation . . . 8

2 Making Huge Page Allocation Feasible 9

2.1 Introduction . . . 9
2.2 Background on Physical Memory Management . . . 11
2.2.1 Unmovable pages . . . 11
2.2.2 Memory allocation . . . 13
2.2.3 RCU and deferred objects . . . 13
2.2.4 Fragmentation mitigation techniques . . . 13
2.2.4.1 Anti-fragmentation . . . 14
2.2.4.2 Defragmentation using memory compaction . . . 14

2.3 A detailed analysis of fragmentation . . . 15
2.3.1 Memory allocation . . . 15
2.3.2 The invisibility of hybrid pageblocks . . . 16
2.3.2.1 Fragmentation via pollution . . . 16
2.3.2.2 LIU migration . . . 17
2.3.2.3 Experimental analysis of fragmentation . . . 18
2.3.3 Delayed reclamation of deferred objects . . . 18
2.3.4 Large memory large problems . . . 19
2.3.5 Impact of fragmentation in virtualized systems . . . 21
2.4 Understanding and addressing the root cause . . . 21
2.4.1 Augmenting fragmentation with unmovability . . . 22
2.4.2 Making operating system responsibilities explicit . . . 23
2.5 Illuminator: Design and Implementation . . . 23
2.5.1 Explicit management of hybrid pageblocks . . . 24
2.5.1.1 Minimizing UI by mitigating fragmentation via pollution . . . 24
2.5.1.2 Eliminating LIU migration . . . 25
2.5.2 Reclaiming pageblocks from the hybrid pool . . . 25
2.5.3 Eliminating susceptibility to page locations . . . 26
2.5.4 Timely reclamation of deferred objects . . . 26
2.5.5 Implementation notes . . . 27
2.6 Evaluation . . . 28
2.6.1 Experimental setup and workloads . . . 28
2.6.2 The cost model for memory compaction . . . 28
2.6.3 Huge page allocations with `stress-highalloc` . . . 29
2.6.4 Performance results on bare-metal . . . 30
2.6.4.1 Overall performance improvement . . . 31
2.6.4.2 Latency and OS jitter . . . 33
2.6.4.3 Performance isolation . . . 33
2.6.5 Performance in virtualized environments . . . 35
2.7 Discussion . . . 36
2.8 Summary . . . 37

3 Fine-grained Huge Page Management **38**
3.1 Introduction . . . 38
3.2 Motivation . . . 40

3.2.1 Address translation overhead vs. memory bloat . . . 40
3.2.2 Page fault latency vs. number of page faults . . . 42
3.2.3 Huge page allocation across multiple processes . . . 43
3.2.4 How to capture address translation overheads? . . . 45
3.3 Design and Implementation . . . 46
3.3.1 Asynchronous page pre-zeroing . . . 47
3.3.2 Managing memory bloat vs. address translation performance . . . 48
3.3.3 Fine-grained huge page promotion . . . 50
3.3.4 Huge page allocation across multiple processes . . . 52
3.3.5 Limitations and discussion . . . 52
3.4 Evaluation . . . 53
3.4.1 Performance advantages of fine-grained huge page promotion . . . 54
3.4.2 Fairness advantages of fine-grained huge page promotion . . . 58
3.4.3 Performance in virtualized systems . . . 65
3.4.4 Bloat vs. performance . . . 65
3.4.5 Fast page faults with async page pre-zeroing . . . 67
3.4.6 Performance overheads of HawkEye . . . 68
3.4.7 Comparison between Hawk-PMU and HawkEye-G . . . 70
3.5 Summary . . . 70

4 Leveraging Architectural Support for All Page Sizes 71
4.1 Introduction . . . 71
4.2 Methodology . . . 74
4.3 How useful are 1GB large pages? . . . 75
4.3.1 1GB pages in native execution . . . 75
4.3.2 1GB pages under virtualized execution . . . 77
4.3.3 Importance of using all large page sizes . . . 78
4.4 Trident: Dynamic allocation of all page sizes . . . 80
4.4.1 Design and implementation . . . 81
4.4.1.1 Managing 1GB physical memory chunks . . . 81
4.4.1.2 Allocating large pages during page fault . . . 81
4.4.1.3 Large page promotion . . . 82
4.4.2 Smart compaction . . . 85
4.5 Trident^{pv}: Paravirtualizing Trident . . . 87
4.6 Evaluation . . . 89

4.6.1 Performance evaluation on bare-metal systems . . . 89
4.6.2 Evaluating Trident's design components . . . 92
4.6.3 Performance under virtualization . . . 94
4.7 Summary . . . 96

5 Mitigating NUMA Effect on Address Translation 97
5.1 Introduction . . . 97
5.2 Analysis of 2D page table placement . . . 100
5.2.1 Analysis of thin workloads . . . 101
5.2.2 Analysis of wide workloads . . . 103
5.3 vMitosis: Design and Implementation . . . 105
5.3.1 Design overview . . . 106
5.3.2 Page table migration . . . 107
5.3.2.1 Page table migration in NV (NUMA-visible) configuration . . . 107
5.3.2.2 Page table migration in NUMA-oblivious NO-P and NO-F configurations . . . 108
5.3.2.3 Linux/KVM implementation . . . 108
5.3.3 Page table replication . . . 109
5.3.3.1 ePT Replication . . . 109
5.3.3.2 gPT replication in NV (NUMA-visible) configuration . . . 110
5.3.3.3 gPT replication in NO-P (NUMA-oblivious paravirtualized) configuration . . . 111
5.3.3.4 gPT replication in NO-F (NUMA-oblivious fully virtualized) configuration . . . 111
5.3.3.5 Linux/KVM implementation . . . 113
5.3.4 Deploying vMitosis . . . 113
5.4 Evaluation . . . 114
5.4.1 Evaluation with page table migration . . . 114
5.4.2 Evaluation with page table replication . . . 117
5.4.2.1 Page table replication in a NUMA-visible scenario . . . 117
5.4.2.2 Page table replication in a NUMA-oblivious scenario . . . 119
5.4.3 Replication vs. migration of page tables . . . 121
5.4.4 Memory and runtime overhead of vMitosis . . . 123
5.4.5 Summary of results . . . 125
5.5 Discussion . . . 125

5.5.1 Huge (large) pages 125
5.5.2 Shadow page tables 126
5.6 Summary 127

6 Related work **128**

7 Conclusion and Looking Forward **134**

Bibliography **136**

List of Figures

1.1 The toal volume of data or information created, captured, copied and consumed worldwide (source: [168]). 2

1.2 Example of a page table walk on x86 with 4KB pages that require 4-level page tables. PFN (page frame number) represents the physical base address of a 4KB page. With 2MB and 1GB pages, PMD and PUD directly provide the physical base address of the data page, respectively. 4

2.1 Virtual-to-physical memory mappings. 11

2.2 Physical memory allocation in Linux. The slab allocator allocates kernel objects while the buddy allocator serves pages to the slab allocator (from the unmovable pool) and to the user space (from the movable pool). 12

2.3 Two-way classification based anti-fragmentation leads to fragmentation via pollution because the buddy allocator cannot reuse hybrid pageblocks during fallbacks. For example, pollution of P1 can be avoided by reusing P2 during C→D. For clarity, hybrid pageblocks are colored yellow in this figure, but the Linux kernel treats them as either red or green, depending on which pool they belong to. 16

2.4 Rate of pageblock pollution with a synthetic benchmark that repeatedly stress the buddy and the slab allocator simultaneously. 19

2.5 Execution time of `milc` with no compaction (none), with asynchronous compaction (async) and with synchronous and asynchronous compaction (sync+async) at 0.75 unmovability index. 20

2.6 Linux creates avoidable fragmentation since its unmovability index is much higher than a perfect page-clustering algorithm. Illuminator is very close to an ideal system. 22

2.7 Illuminator explicitly manages hybrid pageblocks in a different pool to prevent fragmentation via pollution. Prudence helps Illuminator by minimizing callbacks to `alloc_pages` with timely reclamation of deferred objects. 23

2.8 Explicit management of hybrid pageblocks improves page clustering. Once P2 is yielded to the hybrid pool during A→B, it is reused until state K. P1 is added to the hybrid pool only when P2 fails to allocate memory. 24
2.9 The location of unmovable pages affects the outcome of compaction in the two-way classification approach. In this case, Linux can allocate a huge page only in scenario A while Illuminator can allocate huge page in both A and B. 26
2.10 Performance relative to baseline pages at 0.25 (Linux-M), 0.5 (Linux-H) and 0.75 (Linux-C) unmovability indices. Illuminator's performance is presented once which is valid for all fragmentation indices considered. Notice that the performance in Linux degrades as fragmentation increases resulting in worse than the performance of baseline pages at 0.75 unmovability index for most applications. 30
2.11 Reduction in the cost of compaction with Illuminator at different fragmentation levels. 32
2.12 Maximum latency for `MySQL` read requests from 10 iterations at high fragmentation level (UI=0.5). 34
2.13 Slowdown for applications (lower is better) while running alongside `milc` at high fragmentation. 34
2.14 Performance improvement over Linux-H (UI=0.5) in a virtualized system when Illuminator is deployed at host, guest and both. 35

3.1 Resident Set Size (RSS) of `Redis` server across 3 phases: P1 (insert), P2 (delete) and P3 (insert). 41
3.2 Our design objectives in HawkEye. 47
3.3 Average offset to the first non-zero byte in baseline (4KB) pages. First four bars represent the average of all workloads in the respective benchmark suite. 50
3.4 A sample representation of `access_map` for three processes A, B and C. 51
3.5 Performance speedup (top sub-figure) and time saved per huge page promotion (bottom sub-figure) over baseline pages. 54
3.6 Access-coverage in application virtual address space, MMU overhead and the number of huge page promotions for `Graph500`. HawkEye reduces MMU overhead much faster and with fewer huge pages than Linux and Ingens. 55
3.7 Access-coverage in application virtual address space, MMU overhead and the number of huge page promotions for `XSBench`. HawkEye reduces MMU overhead much faster and with fewer huge pages than Linux and Ingens. 56

3.8 MMU overheads of three identical instances of `Graph500` while running simultaneously. Linux allocates huge pages in the order of process creation, and therefore MMU overheads reduce in the same order. Ingens allocates huge pages fairly but takes longer to reduce MMU overhead as it allocates many huge pages in TLB insensitive regions for these applications (low virtual addresses). HawkEye uses hardware performance counters and access-coverage based huge page promotion and is therefore more efficient than Linux and Ingens. 59
3.9 Number of huge pages promoted for all three instances of `Graph500` over time. Linux promotes huge pages in the order of process creation (e.g., `Graph500-1` followed by `Graph500-2`, and so on). Ingens promotes huge pages to all instances at the same rate (overlapping lines not visible in the graph). HawkEye also promotes huge pages at roughly similar rate to all instances, but based on the estimated benefits of allocation. 60
3.10 MMU overheads of three identical instances of `XSBench` while running simultaneously. Linux and Ingens takes a long time to reach the most TLB sensitive regions (high virtual addresses). Therefore, they are unable to reduce MMU overheads in this case. HawkEye uses hardware performance counters and access-coverage based huge page promotion and is therefore more efficient than Linux and Ingens. 61
3.11 Number of huge pages promoted for all three instances of `XSBench` over time. Linux promotes huge pages in the order of process creation. Ingens promotes huge pages to all instances at the same rate. HawkEye also promotes huge pages at roughly similar rate to all instances, but based on the estimated benefits of huge pages (note that overlapping lines are not clearly visible for Ingens and HawkEye-G. 62
3.12 Performance speedup over baseline pages of TLB sensitive applications when they are executed alongside a lightly loaded `Redis` key-value store in different orders. 64
3.13 Performance compared to Linux in a virtualized system when HawkEye is applied at the host, guest and both layers. 66
3.14 Performance normalized to the case of no ballooning in an overcommitted virtualized system. 68
3.15 Performance overhead of async pre-zeroing with and without caching instructions. The first two bars (NPB and Parsec) represent the average of all workloads in the respective benchmark suite. 69

4.1 Performance impact of different page sizes under native execution. Applications in shade benefit from 1GB pages. 76
4.2 Performance impact of different page sizes under virtualization. Applications in shade benefit from 1GB pages. 77
4.3 Total memory mappable with different page sizes. 79
4.4 Relative TLB-miss frequency. 79
4.5 Trident's large-page promotion algorithm. 83
4.6 Linux's normal compaction (top) and Trident's smart-compaction (bottom). . . 85
4.7 Reduction in the number of bytes copied by smart-compaction. 87
4.8 Traditional copy-based versus Trident^{pv}'s copy-less page promotion. 88
4.9 Performance under no fragmentation. 90
4.10 Performance under fragmentation. 91
4.11 Performance analysis of different components of Trident. 93
4.12 Performance under virtualization. 95
4.13 Trident^{pv}'s performance under fragmented gPA. 95

5.1 Performance impact of gPT and ePT placement configurations on Thin workloads. Details of the configurations are discussed in Table 5.3. 102
5.2 Analysis of 2D page table walk of Wide workloads on NUMA-visible and NUMA-oblivious VMs on a 4-socket machine. Bar for each socket (represented by the number) shows the fraction of 2D page table walks that results in Local-Local, Local-Remote, Remote-Local or Remote-Remote leaf PTE access in gPT and ePT, when TLB misses are serviced for one of the threads running on that socket. 104
5.3 Workload performance with and without ePT and gPT migration. Bars are normalized to base case (LL). Absolute runtime for the base case in brackets. Numbers at the top show speedup with vMitosis over the worst-case setting (RRI). 115
5.4 NUMA-visible: Workload performance with and without vMitosis, normalized to the base case (F). Runtime (in seconds) for the base case are in brackets. Numbers at the top show speedup with vMitosis over the corresponding memory allocation policy of Linux/KVM. 118
5.5 NUMA-oblivious: Workload performance, normalized to the base case (OF). Runtime for the base case in brackets. Numbers on top of the bars show speedup with vMitosis wherever significant. Configuration details are listed in the table at the top. 120

5.6 Throughput of a Thin `Memcached` instance before, during and after migration. In the NUMA-visible case (a), the guest OS migrates `Memcached`. In the NUMA-oblivious case (b), the hypervisor migrates `Memcached`'s VM. 122

List of Tables

1.1 Number of entries for each page size in L1 dTLB and L2 TLB across different generations of Intel x86 systems. Resource allocation for huge pages is on the rise particularly in the L2 TLB. 6

2.1 Distribution of unmovable pages. Illuminator produces only about 5% hybrid pageblocks compared to Linux. 18
2.2 Summary of workloads evaluated. 27
2.3 Software counters used to measure the cost of memory compaction. 28
2.4 Cost of each activity in the Linux kernel. We take W_i to be 20. However, the cost of compaction is not heavily dependent on its exact value. 29
2.5 Huge page allocation success rate for `stress-highalloc` which tries to allocate 90% of memory as huge pages. 29
2.6 Number of huge pages allocated/promoted. Allocation happens in the page fault handler while promotion is done by the `khugepaged` kernel thread in background. 31
2.7 Kernel mode execution time (in seconds) and the percentage of total time spent in the kernel mode for `milc`. 33

3.1 Page faults, allocation latency and performance for a microbenchmark with ≈100GB memory allocation. 43
3.2 Number of TLB sensitive applications in popular benchmark suites. We consider an application to be TLB sensitive if its address translation overhead is more than 3%. 45
3.3 Memory characteristics i.e., resident set size (RSS), working set size (WSS), address translation overheads and speedup huge pages provide over base pages for NPB workloads. % cycles denote the fraction of total CPU cycles spend in address translation. 46
3.4 Methodology used to measure MMU Overhead [114]. 46

3.5 Execution time of 3 instances of `Graph500` and `XSBench` when executed simultaneously. Values in parentheses represent speedup over baseline pages. 58
3.6 Experimental setup for configurations used to evaluate a virtualized system. . . 65
3.7 Memory consumption and throughput of `Redis` key-value store with different huge page management systems. 66
3.8 Performance implications of asynchronous page zeroing. Values for `Redis` represent throughput (higher is better); all other values represent time in seconds (lower is better). 67
3.9 Comparison between HawkEye-PMU and HawkEye-G for two sets of workloads. Values in parentheses represent speedup over baseline pages (whereever significant). 70

4.1 Specification of the experimental system 74
4.2 Specifications of the benchmarks. 74
4.3 Comparison of 1GB and 2MB pages allocated via different mechanisms employed in Trident (without physical memory fragmentation). 84
4.4 Comparison of 1GB and 2MB pages allocated via different mechanisms employed in Trident (with physical memory fragmentation). 84
4.5 Percentage 1GB memory allocation failures 92
4.6 Tail latency (ms) for Redis and Memcached 92

5.1 NUMA support for page tables in state-of-the-art systems. (*) Replication is possible in Mitosis only if the server's NUMA topology is exposed to the guest OS. 100
5.2 Detailed description of the workloads. 101
5.3 CPU, data, gPT and ePT placement for different configurations. A and B represent two different sockets in the system (e.g., A=0, B=1). "I" represents interference due to a different workload. 102
5.4 Migration and replication of 2D page tables in current state-of-the-art virtualized systems. NV: NUMA-visible, NO-P: NUMA-oblivious-paravirtualized, NO-F: NUMA-oblivious-fully virtualized. 106
5.5 Migration and replication of 2D page tables in vMitosis. NV: NUMA-visible, NO-P: NUMA-oblivious-paravirtualized, NO-F: NUMA-oblivious-fully virtualized. 106
5.6 Time to transfer a cache line (in ns) between different vCPU pairs. Bold underlined entries represent vCPU pairs wherein both vCPUs are scheduled on the same NUMA socket. The table is shown partially from the 192x192 matrix we profiled on our system. 112

5.7 Throughput (measured as million PTEs updated per second) of different system calls when invoked with different virtual memory region sizes using 4KB mappings. Numbers in parentheses represent throughput normalized to Linux/KVM. 124
5.8 Memory footprint of 2D page tables for a 1.5TB workload using 4KB pages with different replication factors. Numbers in parentheses represent memory consumption of page tables as a fraction of workload size. 124

Chapter 1

Introduction

The virtual memory abstraction has enabled the software ecosystem to flourish unhindered by decoupling software's view of memory from that of the hardware. Thanks to virtual memory, a programmer can write and compile software once on her computer while the resultant software binary can run on many computers with possibly very different amounts of physical memory. Numerous benefits of virtual memory make it an indispensable part of the computing stack today. However, virtual memory is not a free lunch. The mounting overheads of virtual memory abstraction in the era of data-centric computing made us wonder what we can do to moderate its overhead without sacrificing its benefits while also being sensitive to the needs of modern computing, e.g., fairness under multi-tenancy. This thesis explores this broad question from different angles.

1.1 Benefits of virtual memory

Virtual memory provides many key benefits that are hard for today's software stack to sacrifice.

Ease of programming and portability: Virtual memory provides the illusion of a private memory address space (called virtual address space) to each application. Applications perform their computation using virtual addresses that are translated into physical addresses by the hardware and operating system (OS), transparently to the application. This simplifies programming as developers need not worry about managing the physical memory. Further, the same program can be executed on different machines with varying physical memory capacity, simplifying portability.

Fine-grained memory protection: Virtual memory operates at page granularity i.e., the virtual and physical address spaces are divided into page size units. With the help of the OS, the processor provides the ability to use different memory access permissions in different parts

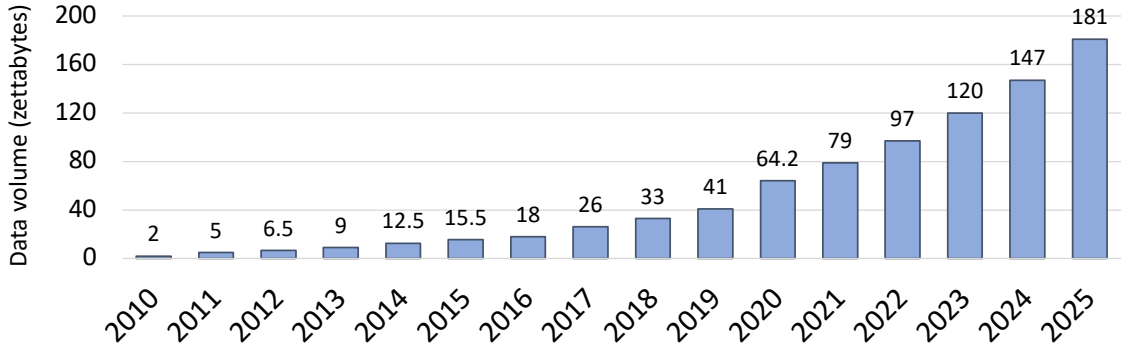


Figure 1.1: The toal volume of data or information created, captured, copied and consumed worldwide (source: [168]).

of the applications (e.g., users can mark some data as read-only).

Resource sharing and isolation: A virtual memory page can be mapped in any physical page frame of the same size. This allows two or more applications to share the same physical server wherein they can use different parts of physical memory. The hardware and OS co-design ensures that a process cannot access unauthorized data of a different process.

Memory over-subscription: Virtual memory allows an application to execute on a machine that provides less physical memory than its requirements. The hardware and OS co-ordinate to support memory over-subscription so that frequently accessed data resides in physical memory while the rest of the data can be stored on disk.

1.2 The cost of virtual memory

The amount of data being generated and consumed is increasing exponentially. Figure 1.1 shows the total volume of data generated and processed, including predictions for upcoming years, for each year between 2010 and 2025. This shows a growth of two orders of magnitude in the volume of data in less than two decades. To transform these enormous amounts of data into actionable information, we need to compute upon it, which implies accessing vast amounts of data from physical memory.

Applications access the data using virtual addresses. Converting an application-generated virtual address into the corresponding physical address where the data resides is a prerequisite for the computation. A processor caches recently used addresses in (often multi-level) hardware structures called Translation Lookaside Buffers (TLBs) to accelerate the virtual-to-physical address translation. If a translation is found in the TLB, there is little virtual-to-physical address translation overhead. However, servicing TLB misses are slow and are the primary source

of address translation overheads. Unfortunately, while the application's memory footprint is increasing at an unprecedented rate, it is not possible to scale up TLB sizes in the same proportion due to fundamental hardware limitations. Consequently, TLB misses and thus the address translation overheads due to virtual memory are sky-rocketing [55, 58, 136].

The size of TLBs cannot be increased arbitrarily due to many fundamental limitations. For example, a TLB is looked up on every load/store instruction. Consequently, even if it has a few entries, TLB's size adds to the power budget and could impact the CPU cycle time. Previous works and industry studies have demonstrated that TLBs could be an energy-hungry part of a processor, and it alone can account for about 6% of total chip power [166, 53, 113] and is known to show up as a hotspot due to frequent lookups [155].

Such limitations leave a large gap between the amount of memory accessible to an application and the amount of address translation coverage provided by the TLBs. In turn, the limited coverage of TLBs entails frequent TLB misses for large memory applications. Further, TLB misses often involve accessing physical memory – adding significant latency to address translation. For example, accessing physical memory on an Intel Xeon Skylake server takes about 250 CPU cycles whereas an L2 TLB access requires only nine cycles. High latency of virtual-to-physical address translation can therefore slow down an application considerably [55, 91, 121]. The overheads further worsen under virtualization as it adds an additional level of address translation for applications executing within virtual machines [136, 91].

The address translation mechanism: The hardware and OS coordinate for virtual-to-physical address translation. The translation occurs at the granularity of a "page": x86 systems use 4KB page size, by default. Here, the OS creates address mappings while allocating and de-allocating physical memory and stores them in per-process page tables.

On x86 systems, the page table is a multi-level radix tree. The number of levels in the tree depends on the page size. For example, if 4KB pages are used, then a page table is a 4-level tree. These levels are referred to as PGD (page global directory), PUD (page upper directory), PMD (page middle directory) and PTE (page table entry).

Current x86 systems represent each virtual address with 48 bits. These 48 bits are divided into five parts during address translation: four parts of nine bits each index in different levels of the page table tree while the remaining 12 bits represent the offset in the 4KB page.

On a context switch, the OS loads the processor's control register $cr3$ with the physical address of the root of the page table tree i.e., PGD. The CPU first looks up a virtual address in the TLB to identify its physical address. Current x86 systems use 2-level TLBs per core, referred to as level-1 (L1) and level-2 (L2), to balance the capacity and look up latency of caching structures. If the address translation is present in either L1 or L2 TLB, the CPU continues

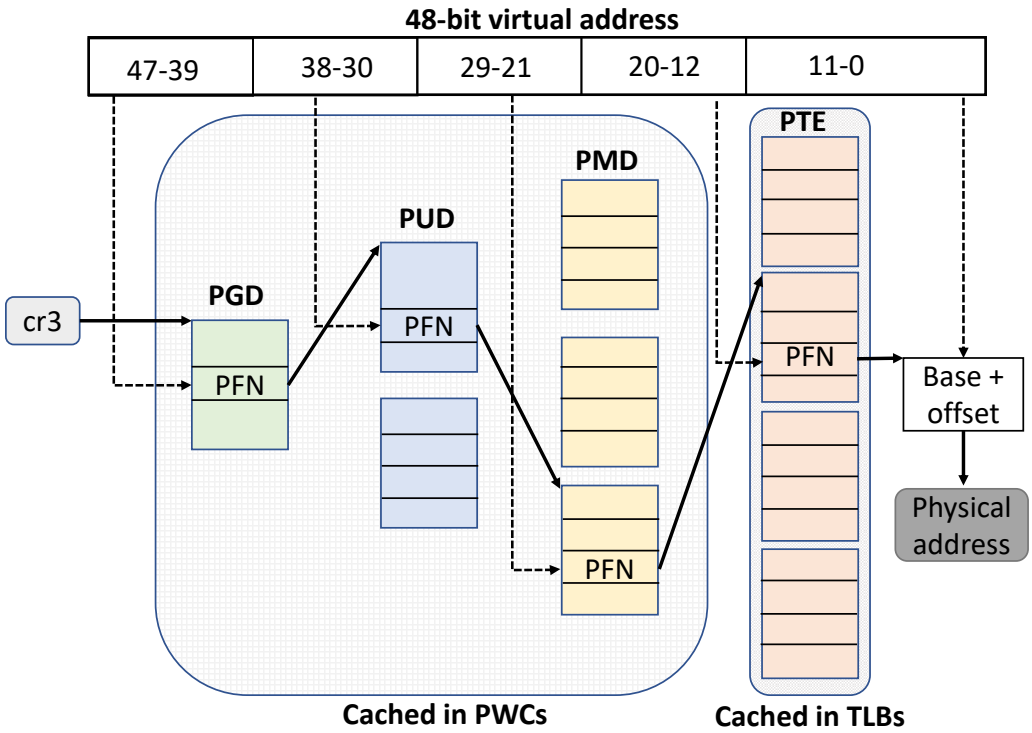


Figure 1.2: Example of a page table walk on x86 with 4KB pages that require 4-level page tables. PFN (page frame number) represents the physical base address of a 4KB page. With 2MB and 1GB pages, PMD and PUD directly provide the physical base address of the data page, respectively.

with regular operations. Otherwise, it initiates a page table walk to fetch the translation from page tables and stores it in the TLB for future references. Figure 1.2 provides an overview of a page table walk on x86 systems using 4-level page tables.

Modern processors employ various mechanisms to reduce the cost of address translation, e.g., x86 systems cache entries from the higher levels of the page tables (PGD, PUD and PMD) in page walk caches (PWCs) [59]. Further, current x86 systems employ two page table walkers per core to hide the effect of long latency address translation. Unfortunately, the overhead of virtual-to-physical address translation continues to be a significant performance concern for data-centric applications. For example, Google recently reported that many of their data-center workloads spend 20-30% of total CPU cycles in address translation alone [106].

Address translation under virtualization: Virtualization adds one more layer of indirection to address translation. Virtualized systems employ two levels of page tables where (1) the guest page tables translate guest virtual addresses (gVA) to guest physical addresses (gPA), and (2) the extended page tables translate gPA to host physical addresses (hPA) [58]. The

TLB stores a direct mapping from gVA to hPA for fast translation. On a TLB miss, the hardware page table walker needs to do a 2D traversal of both page tables to resolve the missing translation. This results in up to 24 memory accesses with four level page tables compared to only up to 4 without virtualization [92]. Therefore, virtual-to-physical address translation is significantly more expensive under virtualization [92, 91, 136].

Effect of non-uniform memory access latency on address translation: Memory access latency is often non-uniform on modern hardware. Large servers are often built using multi-socket architectures where multiple CPUs are connected together and physical memory is distributed across all CPUs [63, 78, 123]. Each CPU can access its local memory much faster in such systems than accessing memory attached to a remote socket. On our experimental 4-socket Intel Xeon Cascade Lake server, accessing a remote memory object takes around 140ns, whereas local memory access latency is about 90ns. The effect of non-uniform memory access (NUMA) latency also affects the virtual-to-physical address translation because a page table walk may involve one or more high latency accesses [39].

1.3 Problem statement

The overhead of virtual-to-physical address translation is already a big performance concern. These overheads are likely to increase in the future. For example, the advent of denser non-volatile memory technologies promises to increase physical memory capacity significantly [111, 137]. This would further widen the gap between TLB coverage and physical memory size. Further, future systems would require deeper page tables to support large address spaces. Notably, the Linux kernel has already incorporated 5-level page tables in its memory management subsystem [126]. Deeper page tables would therefore require more physical memory accesses for address translation e.g., up to 35 with 5-level 2D page tables [108]. Therefore, the frequency of TLB misses, and the cost of each TLB miss would be considerably higher in future systems.

In this dissertation, our objective is to moderate the overhead of virtual-to-physical address translation. We set out towards this goal while also avoiding modifications to application code and avoiding additional hardware. Any enhancement requiring application modification faces the programmer's inertia to modify "working" code. Hardware changes take time to deploy and often incur significant design costs and complexity.

To achieve these objectives simultaneously, a significant part of this dissertation focuses on maximizing the benefits of "huge pages" – an essential hardware feature designed to limit address translation overheads. Huge pages are widely supported by all major processors today.

	L1 dTLB			L2 TLB		
	4KB	2MB	1GB	4KB	2MB	1GB
Sandy Bridge (2011)	64	32	4	512	0	0
Haswell (2013)	64	32	4	1024	1024	0
Skylake (2015)	64	32	4	1536	1536	16
Ice Lake (2021)	64	32	8	2048	1024	1024

Table 1.1: Number of entries for each page size in L1 dTLB and L2 TLB across different generations of Intel x86 systems. Resource allocation for huge pages is on the rise particularly in the L2 TLB.

Introduction to huge pages: Huge pages increase the granularity of address translation, in turn improving the performance of virtual memory in two ways. We explain these benefits using the example of 2MB and 1GB pages available on modern x86 systems. First, a huge page entry maps a much larger portion of virtual address space in the TLB e.g., 2MB or 1GB. This significantly increases the coverage of TLBs and reduces TLB misses. Second, huge pages need smaller page tables. For example, 2MB and 1GB huge pages require only three and two levels in the page tables, respectively. This reduces the number of memory accesses involved in a page table walk. Huge pages are also referred to as large pages and superpages. In this dissertation, we use these terms interchangeably.

Table 1.1 shows that hardware vendors have invested significant resources in huge pages over the years. For example, L2 TLB did not support huge pages in the Sandy Bridge micro-architecture launched in 2011 whereas, recently launched Ice Lake systems provide 1024 entries each for 2MB and 1GB pages in the L2 TLB. Therefore, it is evident that processor vendors rely on huge pages to minimize address translation overheads. Unfortunately, lack of adequate memory management algorithms in current operating systems and hypervisors has often led to disappointing performance results with huge pages in production deployments [7, 8, 12, 17, 2, 75, 72]. This dissertation makes several contributions in highlighting and addressing the challenges involved in huge page management.

1.4 Contributions of this dissertation

Our contributions span across four systems: Illuminator, HawkEye, Trident and vMitosis [144, 145, 146, 156]. We provide a brief summary of these contributions below:

Illuminator: We bring forth subtle interactions between external physical memory fragmentation and huge pages. External fragmentation leaves free memory in non-contiguous blocks. Since each huge page must be mapped in contiguous physical memory, external fragmentation makes allocating huge pages hard for an OS. Unfortunately, recovering from fragmentation is

a more complex challenge for an OS because some physical pages cannot be migrated, e.g., those occupied by the kernel's own objects like page tables, inodes, etc. Therefore, such pages (referred to as unmovable pages in the rest of this dissertation) can permanently fragment the system and make it impossible to recover contiguous physical memory.

We show that despite proactive measures employed in the memory management subsystem of Linux, unmovable kernel pages can still deny huge pages to user applications. In a long-running system, unmovable pages cause high de-fragmentation overheads due to excessive page migration. Over time, their effects manifest in performance regressions, OS jitter, and latency spikes. Iluminator clusters kernel objects in a subset of physical memory regions and makes it feasible to allocate huge pages.

HawkEye: In this work, we deal with OS-based huge page management policies that need to balance complex trade-offs between TLB misses, memory bloat, latency, and the number of page faults. These trade-offs appear because while huge pages can minimize TLB misses and the number of page faults, they can increase memory footprint and the latency of page allocation. Therefore, the OS needs to balance their benefits against undesirable memory footprint and latency effects.

In addition, we discuss the performance and fairness issues that appear with external memory fragmentation. Fragmentation creates a mismatch between the application demand for huge pages and the system's supply of contiguous physical memory. Therefore, the OS is responsible for identifying virtual address regions that can benefit the most from huge pages. In such cases, judicious allocation of huge pages is also essential for fairness, e.g., when multiple tenants share the same physical server.

In HawkEye, we propose asynchronous page pre-zeroing to simultaneously optimize for low latency and few page faults. We propose automatic bloat recovery to effectively deal with the trade-offs between TLB misses and memory bloat at runtime. HawkEye addresses the performance and fairness challenges by allocating huge pages based on their estimated profitability. We derive these estimations from hardware performance counters and by profiling the memory access patterns through page tables.

Trident: Illuminator and HawkEye try to maximize the value of 2MB huge pages. However, recent findings have shown that even after employing 2MB pages, more than 20% CPU cycles are wasted in handling TLB misses for data center applications [106]. We notice that this problem can be addressed by using 1GB huge pages e.g., recent x86 IceLake systems provide 1TB per-core TLB coverage via 1GB pages. Unfortunately, using 1GB pages in current system designs requires prior reservation of physical memory and explicit hints from the applications.

These methods are unsuitable for legacy applications and dynamic workload environments where prior reservation of physical memory can make a significant part of physical memory unavailable to the rest of the system.

To overcome this limitation of current systems, we propose a multi-level huge page management framework called Trident that judiciously allocates 1GB, 2MB and 4KB pages, transparently to the applications, as deemed suitable at runtime.

vMitosis: In this work, we first show that despite current operating systems and hypervisors using sophisticated NUMA memory management techniques, page table walks still need to access remote memory. These high latency memory accesses delay address translation and considerably slow down applications. The effect of non-uniform memory access latency is more pronounced in virtualized systems where nested page table walks require a higher number of memory accesses than native systems. Interestingly, the slow down observed due to remote page table accesses can even outweigh that of accessing remote data, even though page tables consume less than 1% memory of overall application footprint.

vMitosis mitigates the effect of NUMA on page table walks by enabling each core to handle TLB misses from its local socket. We achieve this by judiciously migrating and replicating page tables across NUMA sockets.

We have implemented our proposed systems in the Linux OS kernel and KVM hypervisor. Our optimizations are transparent to the users, and using them does not require any hardware or application modifications. Our artifacts are publicly available [27, 28, 29, 30].

1.5 Organization of this dissertation

The rest of this dissertation is organized as follows. Chapter 2 presents a comprehensive study of the interaction between huge pages and physical memory fragmentation and how we address fragmentation with Illuminator. In Chapter 3, we discuss various other challenges involved in the management of huge pages and how we address them with HawkEye. Chapter 4 details the motivation for harnessing hardware support for all page sizes, along with the design and implementation of Trident. In Chapter 5, we provide a detailed analysis to expose the effect of non-uniform memory accesses on address translation. We also present the design and implementation of vMitosis to address these challenges. In Chapter 6, we discuss prior works related to the mitigation of virtual-to-physical address translation overheads and associated memory management challenges. We also highlight the key differences between the contributions of this dissertation and prior work. Finally, we conclude in Chapter 7 and discuss some promising directions for future work.

Chapter 2

Making Huge Page Allocation Feasible

In Chapter 1, we discussed how the increasing cost of virtual-to-physical address translation has inspired support for huge pages in modern processors. In theory, huge pages can greatly reduce address translation overhead by minimizing TLB misses and reducing the number of memory accesses required during page table walks. However, huge pages must be mapped in contiguous physical memory. Unfortunately, physical memory gets fragmented into smaller non-contiguous blocks which makes it difficult for an OS to allocate huge pages.

In this chapter, we first discuss how the lack of adequate fragmentation mitigation strategies affect huge page performance in current systems. We then propose a system called Illuminator to address the shortcomings of current systems.

Note that, in this chapter, fragmentation refers to external memory fragmentation – a phenomenon that is responsible for creating many small non-contiguous free memory blocks. Fragmentation also appears in another form as "internal" fragmentation that refers to act of allocating more memory than required. We discuss the interaction of huge pages with internal fragmentation in Chapter 3.

2.1 Introduction

We find that the profitability of huge pages depends on the state of physical memory fragmentation as huge pages must be mapped in contiguous memory [96, 139]. In long-running systems, unfavorable fragmentation can (and often does) become a source of poor performance.

Users have repeatedly experienced performance degradation, high kernel space CPU utilization and latency spikes while using huge pages with important applications such as Hadoop, MongoDB, Redis and VoltDB [2, 7, 12, 17]. To avoid such issues, most database servers

are shipped with huge page feature being disabled or explicitly recommend users to disable huge pages. Hence, we can safely infer that the huge page support available in hardware for nearly two decades is not effectively utilized by operating systems. Contrary to a popular belief in the academic literature that fragmentation is not a critical problem and operating systems can efficiently recover from fragmentation with memory compaction (i.e., by relocating pages) [55, 91, 151], we show that fragmentation can indeed cause the aforementioned issues with huge pages. Our views are also well corroborated by the Linux kernel community discussions [69, 70].

In fact, fragmentation is one of the most frequently visited problems in the Linux kernel community. We investigated all 36 major Linux kernel releases between version 4.0 (released in April 2015) and 5.15 (released in Oct 2021), and found that 11 kernel releases have incorporated some fragmentation related enhancements [127]. Fragmentation is also a major problem of other operating systems wherein recent reports provide a strong evidence to shows that fragmentation continues to be a primary hindrance to effective huge page management (e.g., Windows [82] and MacOS [173]).

This chapter presents a comprehensive study of the interaction of fragmentation with huge pages in the Linux kernel. We find that the poor handling of unmovable (i.e., kernel) pages, which are found in many operating system designs, is the root cause of many huge page related problems. While previous work indicates that the existing OS designs can effectively handle unmovable pages [116], we believe this to be erroneous at least in the context of long-running systems.

We identify two major issues to be the root cause of various performance anomalies that appear with huge pages: 1) *fragmentation via pollution*, which occurs when memory contiguity is unnecessarily polluted with unmovable pages, and 2) *latency-inducing unsuccessful (LIU) migration*, which occurs when the kernel unnecessarily migrates pages from the polluted regions while attempting to allocate huge pages. While the former often leads to severe fragmentation, the latter induces latency by increasing the cost of recovering from fragmentation. As fragmentation via pollution increases, the overhead of LIU migration starts to dominate the benefits of huge pages. Importantly, both fragmentation via pollution and LIU migration occur due to poor decision making in memory management related subsystems.

With modest changes in the Linux kernel, Illuminator explicitly tracks all unmovable pages to help various subsystems in avoiding unnecessary work. For example, it helps memory compaction in avoiding LIU migration which leads to cost-effective huge page allocations. It also helps the page allocator in efficiently clustering unmovable pages which in turn reduces fragmentation via pollution. These optimizations provide significant performance improvement across

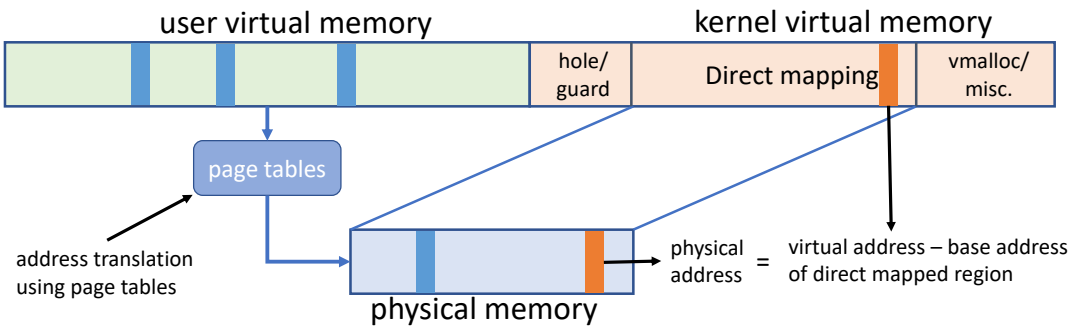


Figure 2.1: Virtual-to-physical memory mappings.

native and virtualized systems.

This chapter of the thesis makes the following major contributions:

- We identify how the existing memory management algorithms lead to various performance anomalies often experienced while using huge pages in the real world (see Section 2.3).
- We present Illuminator, a simple memory management framework that effectively mitigates fragmentation and makes it feasible to allocate huge pages even in stressful conditions (see Section 2.5).
- Through a detailed evaluation, we show that Illuminator significantly outperforms Linux in terms of various performance metrics such as the execution speedup, latency, operating system jitter and performance isolation (see Section 2.6).

2.2 Background on Physical Memory Management

In this section, we discuss some background on physical memory management. While the details discussed in this section are specific to Linux, most concepts are similar across all major operating systems.

2.2.1 Unmovable pages

The mobility (or movability) of a memory page depends on whether all references of the page are tracked or not. In virtual memory systems, the operating system kernel manages user virtual address space with several data stuctures such as page tables and a few other objects (e.g., *vm_mm, vm_area_struct* etc. [64]). Therefore, user memory pages can be migrated by copying their content and updating the corresponding references. Note that this also includes non-pageable *mlocked* memory regions that cannot be swapped out to disk.

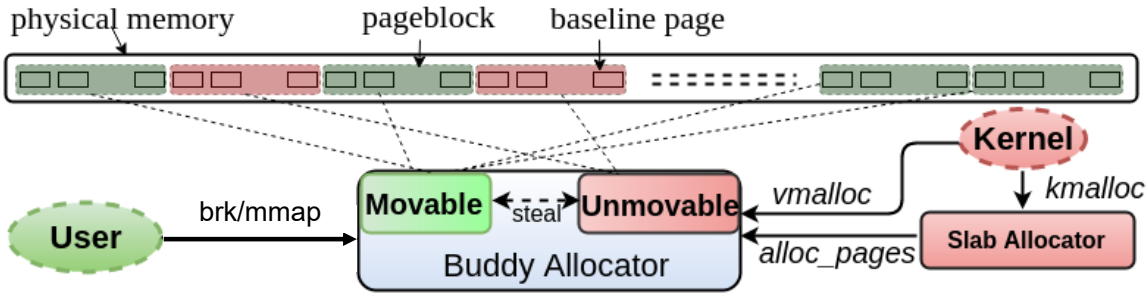


Figure 2.2: Physical memory allocation in Linux. The slab allocator allocates kernel objects while the buddy allocator serves pages to the slab allocator (from the unmovable pool) and to the user space (from the movable pool).

In contrast, there is a one-to-one mapping between (most) kernel virtual addresses and physical memory addresses as shown in Figure 2.1. In other words, physical memory is directly mapped into kernel virtual address space. There are multiple benefits of the direct mapping approach: (1) it allows the kernel to translate a virtual address into physical address using simple BASE+OFFSET arithmetic. This simplifies many kernel operations. For example, the kernel needs to populate page table with the physical memory address when a memory page is allocated to an application. Thanks to direct mapping and simple arithmetic, such operations can be performed faster as compared to using alternate lookup based methods. (2) direct mapping allows the kernel to use the largest page size in its TLB. For example, Linux uses 1GB pages on x86 systems to map its own virtual address space into physical memory. This accelerates address translation when applications execute in the kernel context, and (3) direct mapping simplifies kernel design as developers need not worry about tracking object references. These benefits of direct mapping are well documented in the literature [55, 71].

However, direct mapping has an undesirable side-effect – i.e., direct mapped pages cannot be migrated or swapped to disk because their references are not known. In simple words, direct mapping makes kernel pages unmovable. We find that the unmovability of kernel pages is primary hindrance to effective utilization of huge pages. In fact, as we show in this chapter, unmovable pages can lead to serious performance degradation, if not handled well.

2.2.2 Memory allocation

Figure 2.2 shows the basic memory allocation framework of Linux. Most kernel objects are allocated using the `kmalloc` API. These requests are in turn serviced by the slab allocator. Since most kernel objects are smaller than a regular 4KB page, the slab allocator tries to pack multiple kernel objects in a single physical page, reducing the overall memory consumption of the kernel. To handle allocation requests from different kernel subsystems (e.g., file systems, networking drivers), the slab allocator maintains separate caches – known as slab caches – for different types of object. The slab allocator populates slab caches by requesting physical page frames from the buddy allocator via `alloc_pages` API. Note that page frames occupied by the slab caches are unmovable.

User application requests are serviced at the granularity of pages (e.g., 4KB, 2MB etc.). Therefore, these requests are directly handled by the buddy allocator. If huge pages are enabled, then the kernel preferentially allocates huge pages at the time of page fault, using direct compaction when physical memory is fragmented. If a huge page allocation fails, the kernel tries to allocate a baseline page to allow the application to continue execution. Baseline pages can be promoted to huge pages in the background by the kernel thread called `khugepaged`. For simplicity, we use the term page(s) and baseline page(s) interchangeably.

Current operating systems, including Linux, use a power-of-two buddy allocator that manages free memory in several lists. Lists are ordered based on index wherein each index i contains blocks of 2^i free contiguous 4KB page frames. Linux uses 11 lists ordered from 0 to 10. Hence, the smallest and largest unit of allocation is 4KB and 2MB, respectively.

2.2.3 RCU and deferred objects

Many kernel subsystems relying on the slab allocator employ Read-Copy-Update (RCU) synchronization mechanism to achieve high scalability on multi-core systems. In RCU, writers do not directly update the copy of the object under consideration. Rather, they create a new copy and apply the update operation on the new copy. The old copy is deferred for freeing at a later point in time when it is guaranteed that no thread will reference it. These deferred objects are reclaimed by the synchronization mechanism after a *grace period*, which denotes the completion of all its pre-existing readers. Note that delay in reclaiming deferred objects can increase the kernel memory footprint because slab objects are unmovable by design.

2.2.4 Fragmentation mitigation techniques

Unfavorable placement of unmovable pages can create permanent fragmentation – a situation from which recovering contiguous physical memory becomes difficult or impossible. This can

lead to huge page allocation failures. To prevent these situations, current systems employ a combination of fragmentation avoidance and recovery strategies. In Linux, these strategies are implemented using anti-fragmentation and memory compaction, respectively. We briefly discuss these techniques below.

2.2.4.1 Anti-fragmentation

The purpose of anti-fragmentation is to avoid permanent fragmentation. Permanent fragmentation can be avoided by clustering unmovable pages in a separate pool of memory. To achieve such clustering, the buddy allocator divides physical memory in two disjoint pools (see Figure 2.2): *unmovable* (colored red) and *movable* (colored green). This partitioning is done at pageblock granularity; a pageblock represents a huge page sized contiguous physical memory pool (here, 2MB). The buddy allocator selects a pool based on the source of the allocation request. For example, kernel requests are served from the unmovable pool and user pages are allocated from the movable pool.

The size of movable and unmovable pools is not known to the kernel upfront – it depends on the behavior of workloads that execute at runtime. Enterprise applications like file servers and databases tend to exercise kernel subsystems heavily and therefore consume significant kernel memory. On the other hand, high performance computing applications rarely use kernel services and therefore consume very little kernel memory. Therefore, the size of the movable and unmovable memory pools need to be flexible.

To accommodate the need of dynamic workloads, anti-fragmentation allows physical memory pools to grow and shrink at runtime. Resizing is done using a stealing mechanism wherein one pool can steal pageblocks from the other pool when it runs out of free memory (this event is referred to as a fallback). During a fallback, the color of the stolen pageblock is determined based on the movability status of majority of its children. For example, a pageblock is colored green if majority of its baseline pages are movable. Similarly, a pageblock is colored red if majority of its baseline pages are unmovable.

We note that the actual implementation of anti-fragmentation has more than two pools in the Linux kernel [97, 96]. However, to simplify the discussion in this chapter, we describe it only in the context of two i.e., movable and unmovable pools.

2.2.4.2 Defragmentation using memory compaction

The compaction algorithm recovers memory contiguity by migrating physical page frames in memory. The implementation of compaction algorithm in Linux involves two pointers; from one end, the *migrate scanner* prepares a list of in-use and movable baseline pages while the *freepage scanner* collects free baseline pages from the other end [69]. Pages from the migrate

scanner's list are copied to the freepage scanner's list to recover memory contiguity.

Note that a huge page allocation can fail for two reasons: (1) the amount of total free memory cab be low, and (2) there can be enough free memory but split into multiple small non-contiguous blocks because of fragmentation. Compaction cannot recover memory contiguity in the former case. Therefore, when the kernel fails to allocate a huge page, it first check the Free Memory Fragmentation Index (FMFI) to determine if the allocation is failing due to lack of free memory or due to fragmentation. For a binary buddy allocator [117, 118], FMFI for a particular order of allocation j is calculated as follows (note that j=9 for 2MB huge pages):

$$\boxed{F_i(j) = 1 - \frac{TotalFree/2^j}{BlocksFree}} \tag{2.1}$$

where TotalFree is the number of free baseline pages and BlocksFree is the number of blocks in the buddy allocator among which TotalFree pages are distributed.

If physical memory is not fragmented, then FMFI may be negative. However, FMFI need not be calculated in those cases because allocations can be serviced easily in the absence of fragmentation. If an order j allocation fails, then FMFI lies between 0 (low fragmentation) and 1 (high fragmentation). Therefore, the kernel compacts physical memory if FMFI is beyond a non-negative threshold.

Physical memory can be compacted synchronously or asynchronously by the kernel. Here, synchronous compaction refers to compaction performed in the context of handling a page fault. Synchronous compaction, therefore, is useful as it can help in allocating huge pages eagerly. However, it might affect application latency due to longer page fault handing time. Asynchronous compaction happens out of the critical path of allocation e.g., when the `khugepaged` kernel thread promotes application baseline pages to huge pages in the background.

2.3 A detailed analysis of fragmentation

The virtual memory layer in the Linux kernel has undergone significant changes for accommodating the support for huge pages. However, operations at the physical memory layer (i.e., page/object allocation, compaction) have largely remained unchanged. This section discusses how unnecessary work can occur and become a source of poor performance in long-running systems. First, we introduce the notion of a hybrid pageblock.

2.3.1 Memory allocation

Hybrid pageblock: A pageblock is hybrid if it contains both movable and unmovable pages. We explain how hybrid pageblocks are formed with an example (see Figure 2.3).

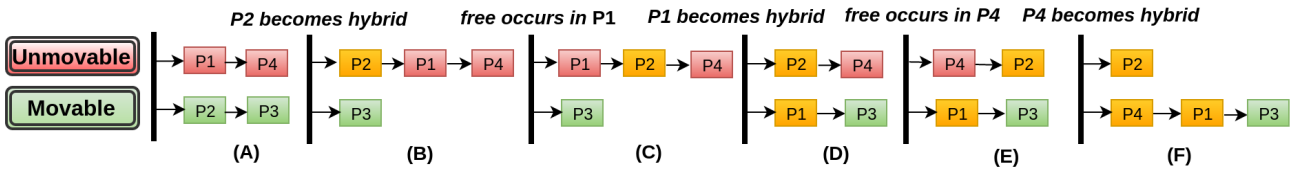


Figure 2.3: Two-way classification based anti-fragmentation leads to fragmentation via pollution because the buddy allocator cannot reuse hybrid pageblocks during fallbacks. For example, pollution of P1 can be avoided by reusing P2 during C→D. For clarity, hybrid pageblocks are colored yellow in this figure, but the Linux kernel treats them as either red or green, depending on which pool they belong to.

Let us assume a system starts with four pageblocks P1 to P4. The two-way classification approach of anti-fragmentation works well until the system reaches state A where P1, P4 belong to the unmovable pool and P2, P3 belong to the movable pool. If P1 and P4 have no free pages left and an unmovable page is requested by the kernel, the algorithm of Linux steals and adds P2 to the unmovable pool during system transition from A→B. P2 thus becomes a hybrid pageblock during A→B, if some movable pages were already allocated from it.

2.3.2 The invisibility of hybrid pageblocks

Two-way classification based anti-fragmentation makes hybrid pageblocks invisible which in turn represents a *stale* view of movability. While, in reality, there are three types of pageblocks (i.e., movable, unmovable and hybrid) in the system, memory subsystems operate on the assumption of only two types of pageblocks – treating each pageblock as either movable or unmovable. This leads to the following scenarios of poor decision making in critical code paths.

2.3.2.1 Fragmentation via pollution

This is a situation where unmovable pages are *unnecessarily* allocated from movable (green) pageblocks. This happens because even though hybrid pageblocks are placed at the head of free lists during fallbacks, they get shifted away from the head over a period of memory allocation and free cycles. Thus, the buddy allocator cannot identify or reuse existing hybrid pageblocks efficiently. For example in Figure 2.3, P1 gets polluted during C→D because the buddy allocator is not aware of P2 already being a hybrid pageblock. Similarly, the transition from E→F also produces a new hybrid pageblock P4. In the worst case, each fallback can produce a fresh hybrid pageblock in the system.

Fragmentation via pollution restricts huge page allocations since a polluted pageblock cannot be allocated as a huge page unless its unmovable pages are freed. In practice, only a

few misplaced unmovable pages can create permanent fragmentation. For example, a 2MB pageblock on x86 systems consists of 512 baseline pages (512×4KB=2MB). However, a single unmovable page is sufficient to pollute a pageblock and hence only 0.19% misplaced unmovable pages (of total system pages) can pollute all the pageblocks (1/512=0.0019) in the worst case. Moreover, the severity of fragmentation via pollution increases along with the size of huge pages — a system using 1GB-sized huge pages can be fragmented by only 0.00038% misplaced unmovable pages.

2.3.2.2 LIU migration

During compaction, the kernel migrates pages from all green pageblocks encountered by the migrate scanner, optimistically believing that a green pageblock can always be emptied. But many hybrid pageblocks are also colored green in the two-way classification of pageblocks (for example, both P1 and P4 are hybrid in state F). This behavior leads to LIU migration.

LIU migration is harmful because it unnecessarily consumes CPU cycles. For example, migrating a baseline page takes about 5 microseconds on our test setup. When a hybrid pageblock contains one unmovable page, the kernel may migrate up to 511 pages from the pageblock adding 2.5 millisecond latency to a huge page allocation. This latency is further exacerbated when many consecutive pageblocks are hybrid. Even worse, huge page allocation can fail even after migrating pages from many hybrid pageblocks.

In addition to copying the page content, compaction also involves other critical operations such as updating the page table(s) and locking a few critical data structures. It also necessitates TLB invalidations because the page being migrated may be mapped in the TLB of some CPU core(s). TLB invalidations are performance using operating system driven inter-processor interrupts (IPIs). The overhead of TLB invalidations is a major performance impediment on large multi-core machines [120, 180]. Therefore, when many pageblocks become hybrid in a system, LIU migration becomes an unnecessary overhead because the kernel migrates pages in response to each huge page allocation request without checking the futility of doing so.

The current kernel design assumes that LIU migration is not an issue as it can prevent long-term fragmentation for sub-pageblock sized allocations. For example, a 64KB allocation request can be served by migrating pages from a hybrid pageblock. It can also free other smaller blocks that can be used to serve future allocations. While we believe such proactive migration is desirable for reducing fragmentation for sub-pageblock allocations, it should not be applicable to huge page allocations because it never produces a free pageblock and hence never reduces long-term fragmentation at the granularity of huge pages.

Number of unmovable pages	Number of hybrid pageblocks	
	Linux (664)	Illuminator (34)
1	24	0
2–50	600	0
50–250	30	0
250–375	10	2
375–512	0	32

Table 2.1: Distribution of unmovable pages. Illuminator produces only about 5% hybrid pageblocks compared to Linux.

2.3.2.3 Experimental analysis of fragmentation

We find that fragmentation via pollution and LIU migration are quite common in Linux. For instance, on a 2GB system which has about 950 pageblocks, the compilation of the kernel source produces 664 hybrid pageblocks with most of the hybrid pageblocks containing a few unmovable pages (see Table 2.1). In fact, 624 out of 664 hybrid pageblocks have less than 50 unmovable pages each. Interestingly, 24 hybrid pageblocks contain one unmovable page each. As discussed above, sparsely polluted pageblocks lead to significant overhead during memory compaction.

2.3.3 Delayed reclamation of deferred objects

In the existing slab-based allocators, deferred objects are not reclaimed immediately after the completion of a grace period for several reasons. For example: 1) RCU manages deferred objects in a queue and reclaims their memory by invoking the registered callback function of each object. Thus the reclamation of objects placed towards the end of the queue is delayed, 2) RCU throttles the rate of reclamation of deferred objects to avoid interfering with applications, irrespective of the state of the slab allocator [134], and 3) the kernel threads responsible for reclaiming deferred objects may be preempted. Until reclaimed, deferred objects cannot be reused as they remain invisible to the slab allocator.

Modern applications that perform thousands of update operations per second generate many deferred objects [161]. The delayed reclamation of deferred objects in turn increases slab consumption. It also leads to high slab churns by forcing the slab allocator to populate slab caches, in response to a new allocation request, even if a lot of deferred objects have waited for more than a grace period. Recall that slab pages are unmovable. Hence, unnecessary calls to `alloc_pages` can increase fragmentation via pollution.

Figure 2.4 shows the impact of fragmentation via pollution with a synthetic benchmark

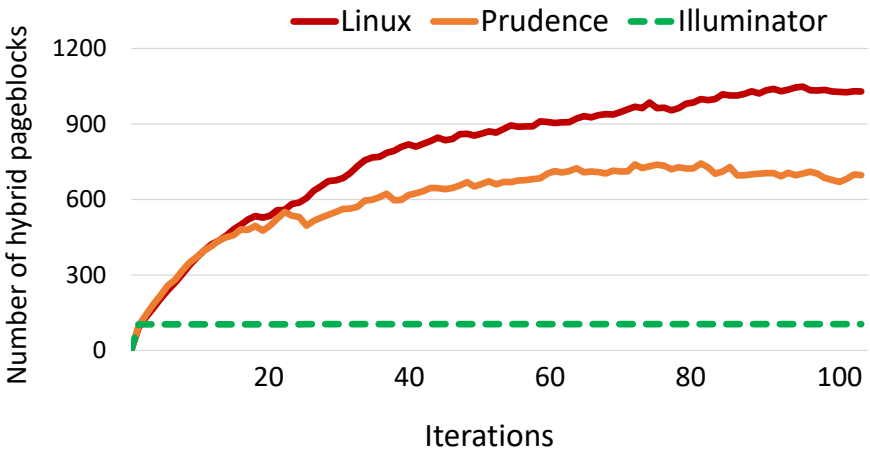


Figure 2.4: Rate of pageblock pollution with a synthetic benchmark that repeatedly stress the buddy and the slab allocator simultaneously.

that we use to fragment the memory. Unlike existing tools, our benchmark stresses the buddy allocator (with anonymous mapping) and the slab allocator (with file create and delete operations) at the same time and can be used to control the level of fragmentation. We execute the benchmark in a loop and observe the following.

In Linux, the delayed reclamation of deferred objects increases the number of calls to `alloc_pages` resulting in 1021 hybrid pageblocks at the end of the hundredth iteration. The recently proposed Prudence dynamic memory allocator [153] reclaims deferred objects immediately after the completion of a grace period which reduces the number of hybrid pageblocks to 691, a 35% reduction compared to Linux. Hence, we replace the slab allocator with Prudence.

2.3.4 Large memory large problems

We also find a counterintuitive side effect of LIU migration i.e., it can make performance worse as the size of memory grows. We observed that when a fixed fraction of total pageblocks become hybrid, the execution time of certain applications increases with the size of physical memory.

To verify this, we measure the performance of `milc` (from SPEC CPU2006) with different memory sizes when 75% of total pageblocks are hybrid. This workload is chosen because it stresses the memory allocation and compaction code paths due to aggressive memory allocation behavior. Note that a large memory system has more hybrid pageblocks and takes more time to reach a state where 75% of total pageblocks are hybrid. However, in this experiment, we do not consider the time taken to fragment the memory; we are interested only in the impact of fragmentation. The amount of LIU migration is also higher on a large memory system as it

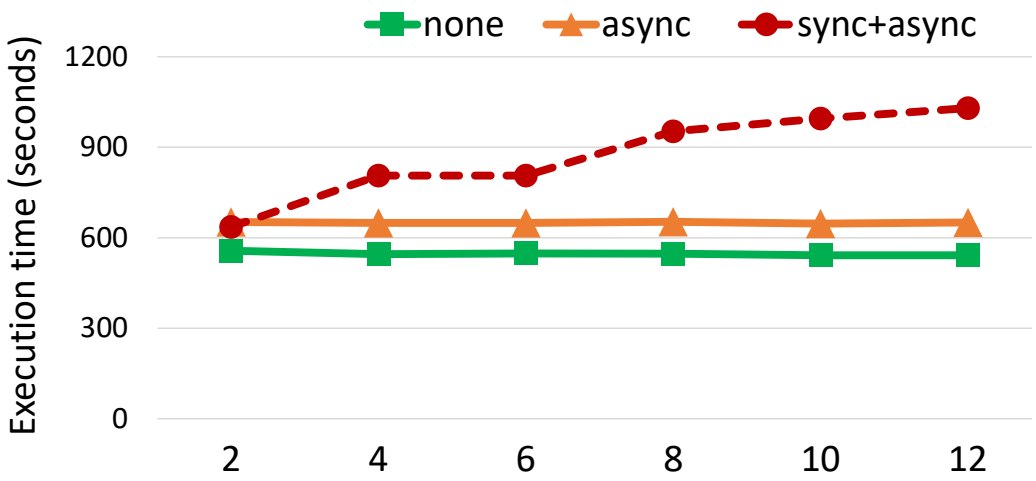


Figure 2.5: Execution time of `milc` with no compaction (none), with asynchronous compaction (async) and with synchronous and asynchronous compaction (sync+async) at 0.75 unmovability index.

contains more hybrid pageblocks. For example, 2GB and 10GB memory systems have about 750 and 3750 hybrid pageblocks in this case (75% of 1k and 5k pageblocks, respectively).

Figure 2.5 shows that `milc` completes in constant time for all memory sizes when compaction is not required i.e., in the non-fragmented case. The execution time is also constant in the fragmented case (but higher than the non-fragmented case) with only asynchronous compaction which is designed to avoid interfering with applications. To the contrary, synchronous compaction stalls applications at the time of page fault. Hence, its impact increases as memory capacity (and hence the number of hybrid pageblocks) grows. As a result, 12GB memory system results in 1.9× higher execution time compared to a 2GB memory system.

This occurs because the kernel controls the rate of only asynchronous compaction, with a configuration parameter whose default value is 1.6MB per second. No such limit is enforced on the rate of synchronous compaction whose cost increases as the number of hybrid pageblocks grow. Hence, applications that allocate memory during their entire execution cycle (e.g., `milc`, `bwaves`, `bzip2` from SPEC CPU2006) are heavily impacted by synchronous compaction. Applications that do not allocate memory frequently are not vulnerable to such behavior. We believe a similar issue may be present in Windows as it prohibits making repeated large page allocations; applications should allocate all large pages one time, at startup [81].

One way to handle this issue is to employ policy-based decisions to also control the rate of synchronous compaction. However, our objective in this chapter is to develop an efficient compaction mechanism that can help in both synchronous and asynchronous code paths.

2.3.5 Impact of fragmentation in virtualized systems

Notice that in virtualized systems, the guest OS (if it is Linux) handles fragmentation with similar algorithms as the host OS. Hence, the severity of fragmentation is higher in virtualized setups because LIU migration takes place in both guest and host when memory is fragmented.

We find that the impact of guest memory fragmentation is more severe than host memory fragmentation as the benefits of huge pages at the guest layer are usually higher than huge pages at the host (for our workloads). However, fragmentation in the host layer also has a significant impact in many cases. Hence, it is important to handle fragmentation in both layers for making the best use of huge pages.

2.4 Understanding and addressing the root cause

In Section 2.3, we discussed how fragmentation via pollution occurs due to unmovable pages, and how it affects system performance because of LIU migration. In this section, we explain why these problems appear in the first place.

The root cause of fragmentation via pollution and LIU migration is the lack of adequate representation of the state of physical memory fragmentation. In particular, the buddy allocator and compaction algorithm are both unaware of hybrid pageblocks.

Let us consider page frame allocation and anti-fragmentation first. The primary objective of anti-fragmentation is to cluster unmovable pages effectively i.e., in as few pageblocks as possible. However, when memory pressure increases in the system, movable and unmovable memory pools frequently steal page frames from each other. This dynamic growth of memory pools inevitably creates many hybrid pageblocks. However, current buddy allocator treats each pageblock as either movable or unmovable while in many cases, majority of the pageblocks are hybrid. Hence, the buddy allocator fails to capture the real movability status of most of the physical memory. In Section 2.5, we show that capturing hybrid pageblocks is invaluable from the clustering perspective.

The problem with compaction is that it relies on FMFI to determine whether physical memory can be defragmented using compaction. However, FMFI does not say anything about unmovable pages. We find that situations are quite common where FMFI is high (indicating that compaction is likely to produce memory contiguity), whereas most of the pageblocks are hybrid – meaning that compaction is unlikely to be successful. Therefore, FMFI fails to capture the difficulty or likelihood of successfully recovering from a fragmented state. Relying on FMFI alone also leads to unnecessary page migration for the same reason.

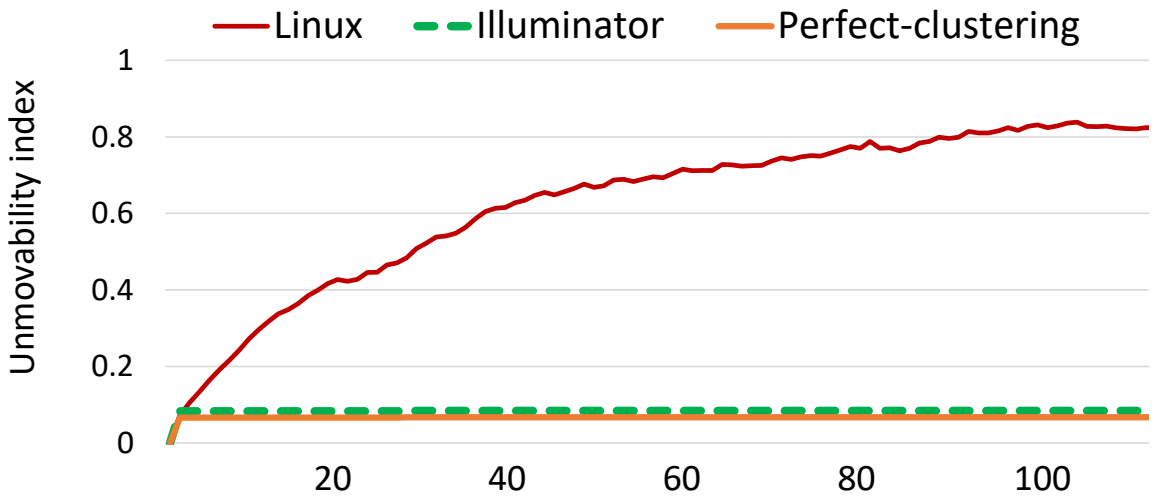


Figure 2.6: Linux creates avoidable fragmentation since its unmovability index is much higher than a perfect page-clustering algorithm. Illuminator is very close to an ideal system.

2.4.1 Augmenting fragmentation with unmovability

Both the issues discussed above can be handled effectively with an enhanced representation of physical memory fragmentation. To do this, we propose a simple metric *unmovable index* (UI), defined as follows:

$$\boxed{UI(j) = \frac{Number\ of\ 2^j\ size\ blocks\ with\ one\ or\ more\ unmovable\ page}{Total\ number\ of\ 2^j\ size\ blocks}} \tag{2.2}$$

UI is defined for each order of allocation. While the definition of UI is generic, in this chapter, we are interested in 2MB huge pages that correspond to j=9. Therefore, UI captures the fraction of pageblocks that are fragmented due to unmovable pages. For example, UI is 0.5 if half of the total pageblocks contain at least one unmovable page each.

Note that the overall state of fragmentation depends on how unmovable pages are spread in physical memory. Sparsely polluted pageblocks result in higher fragmentation because for a fixed amount of unmovable memory, sparse pollution creates more hybrid pageblocks. This is easily captured by UI as many sparsely polluted pageblocks imply a higher value of UI. Also note that UI is not a substitute for FMFI – rather it complements FMFI. In fact, the two-dimensional tuple (FMFI, UI) represents a more accurate state of fragmentation wherein the first element determines how free memory is split across multiple blocks and the second element captures how likely is it for compaction to recover memory contiguity.

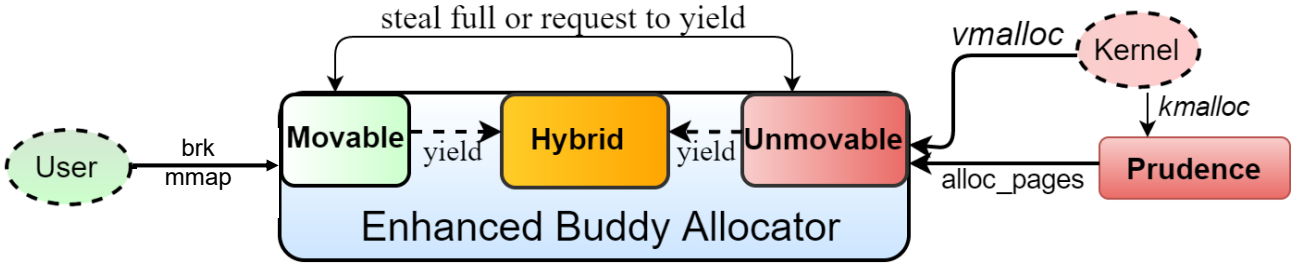


Figure 2.7: Illuminator explicitly manages hybrid pageblocks in a different pool to prevent fragmentation via pollution. Prudence helps Illuminator by minimizing callbacks to `alloc_pages` with timely reclamation of deferred objects.

2.4.2 Making operating system responsibilities explicit

The other advantage of capturing fragmentation as a two-dimensional tuple (FMFI, UI) is that it provides a well-defined metric to the memory management subsystems. It influences the behavior of the buddy allocator and compaction algorithm as follows:

1. The buddy allocator should minimize UI.

2. Compaction should be based on UI and FMFI both.

UI can also be used to capture the behavior of an oracle – a perfect page-clustering algorithm that would produce the minimum possible value for UI by packing all unmovable pages together. Minimum possible UI attainable via an oracle can be defined as follows:

$$\boxed{Minimum\ possible\ UI = \frac{Total\ number\ of\ unmovable\ pages}{Total\ number\ of\ pages\ in\ the\ system}} \tag{2.3}$$

Figure 2.6 shows the same experiment as discussed in Figure 2.4, but in terms of UI. It shows that fragmentation created by Linux is avoidable as the minimum possible value attainable via an oracle is less than 0.1 whereas UI is more than 0.8 in Linux. In Section 2.5, we present the design and implementation of our proposed system Illuminator that uses UI as an important metric to optimize system performance. Figure 2.6 shows that Illuminator is very close to an oracle as its UI is very close to a perfect-clustering algorithm.

2.5 Illuminator: Design and Implementation

To efficiently support huge pages, an OS should mitigate fragmentation via pollution and eliminate LIU migration. Intuitively, any solution that minimizes slab memory consumption is also

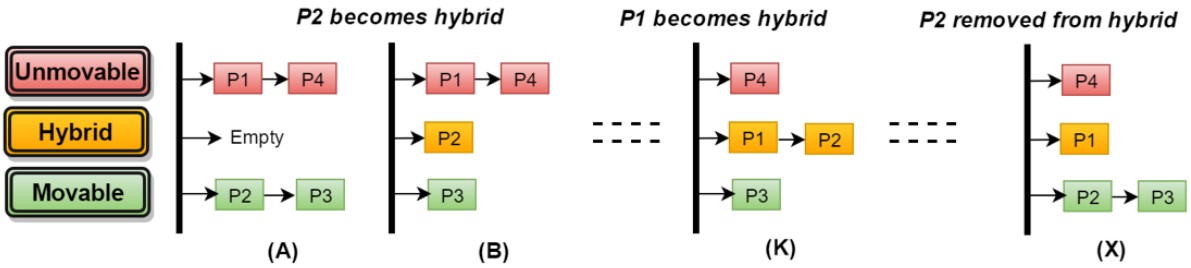


Figure 2.8: Explicit management of hybrid pageblocks improves page clustering. Once P2 is yielded to the hybrid pool during A→B, it is reused until state K. P1 is added to the hybrid pool only when P2 fails to allocate memory.

helpful. Illuminator satisfies these requirements by explicitly managing hybrid pageblocks with a simple design. To optimize the memory consumption of slab caches, we use the Prudence memory allocator [153]. Figure 2.7 represents the high-level design of Illuminator.

2.5.1 Explicit management of hybrid pageblocks

Illuminator partitions memory in three disjoint pools where unmovable and movable pools serve the same purpose as in Linux while the third pool is used to explicitly manage hybrid pageblocks. We represent hybrid pageblocks with *yellow* color. Three-way partitioning based anti-fragmentation provides memory management subsystems a precise view of movability; it guarantees that pageblocks colored red contain only unmovable pages and pageblocks colored green contain only movable pages. This design helps subsystems in making informed decisions.

2.5.1.1 Minimizing UI by mitigating fragmentation via pollution

In Illuminator, the buddy allocator attempts to allocate memory from the corresponding pool first (similar to Linux) but prefers hybrid pageblocks for allocation before a pool is allowed to steal pageblock(s) from the other. If a fallback happens, a pageblock is stolen only if all of its constituent base pages are free. Otherwise, the pageblock is yielded to the hybrid pool and its color is updated to yellow. This way, Illuminator ensures that pageblocks are not polluted unnecessarily. The inclusion of hybrid pageblocks does not add performance overhead in the allocation path as they can be accessed by the buddy allocator in constant time.

We explain the memory allocation flow in Illuminator with an example similar to the one discussed in Section 2.3 (see Figure 2.8). Illuminator behaves similar to Linux until the system reaches state A as the hybrid pool stays empty until this point. If an unmovable page is requested when P1 and P4 have no free pages, the movable pool yields P2 to the hybrid pool and tags it with yellow color. Illuminator guarantees that a new pageblock is not polluted as

long as P2 can successfully serve memory fallbacks. This behavior leads to better clustering of unmovable pages in the long run.

The impact of better page-clustering is shown in Table 2.1. Illuminator reduces the number of hybrid pageblocks to only 34 compared to 664 of Linux, each with many unmovable pages. Note that almost each hybrid pageblock has more than 375 unmovable pages in Illuminator. It reduces the number of hybrid pageblocks by almost 90% for our synthetic benchmark as well (shown in Figure 2.4).

2.5.1.2 Eliminating LIU migration

Illuminator eliminates LIU migration related to huge pages by decoupling huge page related compaction from the compaction induced by smaller (i.e., sub-pageblock) allocation requests. This is achieved by avoiding hybrid pageblocks during huge page allocations. The migrate scanner checks the color of each pageblock before migrating pages and skips the entire pageblock range (i.e., 512 contiguous baseline pages) if the pageblock is hybrid. Similarly, the freepage scanner also skips hybrid pageblocks while collecting free pages.

We deliberately allow the freepage scanner to skip hybrid pageblocks so that free space in the hybrid pool is not exhausted prematurely. If the freepage scanner is allowed to select pages from hybrid pageblocks, then page migration during compaction will quickly exhaust free space in the hybrid pageblocks. This increases the risk of subsequent memory allocations falling back to movable or unmovable pools – which in turn would increase UI. Therefore, skipping hybrid pageblocks is essential to minimize UI and fragmentation via pollution.

2.5.2 Reclaiming pageblocks from the hybrid pool

A hybrid pageblock may become movable, for example, when memory gets freed and the slab allocator returns all its pages to the buddy allocator. It is important to reclaim such pageblocks from the hybrid pool to prevent them from getting polluted by future allocations. Illuminator follows a lazy approach for this task for efficiency; proactively reclaiming pageblocks from the hybrid pool necessitates changes in the fast path of the buddy allocator as it requires keeping track of the number of unmovable pages in each pageblock. In performance sensitive code paths, such accounting is considered to be expensive.

Illuminator reclaims pageblocks from the hybrid pool during compaction. It queries the number of movable and unmovable pages for each hybrid pageblock encountered by either of the two pointers. If a hybrid pageblock is found to contain only movable (or free) pages, Illuminator updates the pageblock color to green and adds it to the movable pool. Similarly, a pageblock is colored red and added to the unmovable pool if it is found to contain only

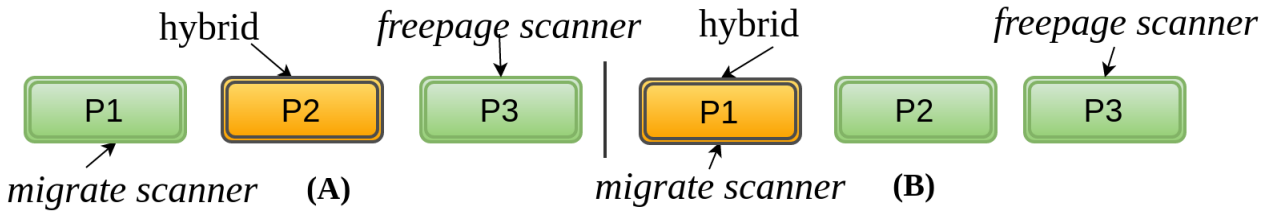


Figure 2.9: The location of unmovable pages affects the outcome of compaction in the two-way classification approach. In this case, Linux can allocate a huge page only in scenario A while Illuminator can allocate huge page in both A and B.

unmovable (or free) pages. If compaction is delayed for a long duration, a thread can be run in the background to periodically scan and reclaim pageblocks from the hybrid pool.

2.5.3 Eliminating susceptibility to page locations

In the Linux kernel, the invisibility of hybrid pageblocks also has an additional side effect i.e., it makes performance susceptible to physical location of unmovable pages. We explain this with an example.

Consider two different scenarios of a system with three pageblocks P1, P2 and P3 along with the positions of the scanning pointers as shown in Figure 2.9. Assume that half of the baseline pages (i.e., 256 pages) are allocated from each pageblock. Also assume that P2 is hybrid in scenario A and P1 is hybrid in scenario B, with one unmovable page each. In Linux, a huge page can be allocated only in scenario A (by migrating P1's pages to P3). In scenario B, LIU migration of P1's pages to P3 would eliminate the possibility of P2 being allocated as a huge page as it will leave P3 with insufficient space to accommodate the pages of P2.

As a consequence of such behavior, the cost of compaction in Linux depends on the spatial distribution of unmovable pages which in turn leads to variable latency in the allocation of huge pages. Illuminator is not susceptible to such behavior as it avoids hybrid pageblocks during compaction.

2.5.4 Timely reclamation of deferred objects

We use Prudence memory allocator to facilitate the timely reclamation of deferred objects. Prudence stores deferred objects in *latent caches* where a latent cache is defined for each slab cache. It also integrates with RCU synchronization mechanism to provide grace period information to the slab allocator. With such information, the slab allocator reclaims deferred objects immediately after the completion of a grace period.

Integration with the synchronization mechanism provides a complete view of in-use, free

and about-to-be-freed objects to the slab allocator. The about-to-be-freed objects also provide crucial hints about the future free operations which are utilized to reduce the footprint of unmovable memory by clustering kernel object allocations within fewer slabs. Together, these optimizations substantially reduce the number of calls to `alloc_pages` API from the slab allocation. For our synthetic benchmark, Prudence creates 691 hybrid pageblocks, a reduction of 33% compared to 1021 of Linux as shown in Figure 2.4.

It is important to highlight that only optimizing the slab allocator is not sufficient because fragmentation via pollution, which is under the control of the buddy allocator, is the primary hindrance to effective mitigation of fragmentation. However, optimizations in the slab allocator complement the explicit management of hybrid pageblocks by reducing the load on the buddy allocator.

2.5.5 Implementation notes

Illuminator is a simple enhancement of allocation and compaction related memory management operations in Linux. It modifies only the infrequently traversed memory fallback path of the buddy allocator with minor changes in the memory compaction code. Prudence memory allocator also affects only the deferred free operations of RCU objects which are executed out of the critical path. Hence, regular fast-path allocation and free operations remain intact in Illuminator. In fact, by eliminating LIU migration, Illuminator makes certain memory management operations simpler and more efficient. For example, it reduces locking overhead on several critical data structures, minimizes cache pollution and requires fewer TLB invalidations. Illuminator is about 1500 lines of code change in the Linux kernel version 4.5 – a change of less than 1.5% of its total memory management code.

Workloads	**Description**
milc, mcf, omnetpp, bzip2	Memory and compute intensive applications from SPEC CPU2006 [105]
mummer, tigr	Genome alignment and sequencing applications from BioBench benchmark suite [43]
CG	Congruent gradient algorithm from NAS Parallel Benchmark Suite (class D) [49]
ferret, vips, canneal, bodytrack, x_264	Compute intensive multi-threaded workloads from PARSEC benchmark suite [62]
PostgreSQL, MySQL	Database servers benchmarked with pgbench [11] and sysbench [14] utilities

Table 2.2: Summary of workloads evaluated.

Terminology	Description
pg_migrate_scanned	Number of pages scanned by the migrate scanner
pg_free_scanned	Number of pages scanned by the freepage scanner
pg_migrate_success	Number of pages migrated during compaction
pg_migrate_failed	Number of pages that failed during migration
pg_isolated	Number of pages that were temporarily removed from the buddy allocator

Table 2.3: Software counters used to measure the cost of memory compaction.

2.6 Evaluation

So far we have discussed how Illuminator mitigates fragmentation via pollution with the help of a few examples such as Table 2.1 and Figure 2.4. This section presents a detailed evaluation of Illuminator in the context of huge pages.

2.6.1 Experimental setup and workloads

Our experimental setup consists of an Intel Ivy-Bridge server with 8 cores running at 2.4GHz with 8MB of Last-Level-Cache and a 500GB SSD drive. L1 dTLB and iTLB contain 64 entries each for 4KB pages and 8 entries each for huge pages. The shared L2 TLB contains 512 entries for 4KB pages but does not support huge pages. We evaluate a wide range of workloads summarized in Table 2.2. Experiments are conducted after disabling swap, to avoid the impact of paging. Experiments that include large memory workloads `CG`, `PostgreSQL` and `MySQL` are conducted with 24GB while other experiments are performed with 8GB physical memory. We use the default huge page promotion rate of 1.6MB per second for the `khugepaged` thread as used in Linux distributions.

2.6.2 The cost model for memory compaction

We use the cost-theoretic model proposed by Gorman to estimate the total cost of memory operations involved in compaction [95]. The model estimates the cost of memory compaction $Cost_{mc}$ in terms of the number of bytes read or written, by tracking system activities and associating each activity with a weight factor (see Table 2.3 and Table 2.4). $Cost_{mc}$ is calculated as follows:

$$\begin{aligned} Cost_{mc} = {} & C_{sm} * pg\_migrate\_scanned + C_{sf} * pg\_free\_scanned + C_i * pg\_isolated \\ & + C_{mc} * pg\_migrate\_success + C_{mf} * pg\_migrate\_failed \end{aligned} \tag{2.4}$$

$Cost_{mc}$ determines the amount of memory traffic induced by the compaction code which

Notation	Description	Value
C_a	Accessing page structure: $sizeof(struct\ page)/word\_size$	8
C_{mc}	Migrate page copy: $(C_a + PAGE\_SIZE/word\_size) \times 2$	1040
C_{sm}	Migrate scanning: C_a, this is equivalent to scanning a page structure	8
C_{sf}	Freepage scanning: C_a, this is equivalent to scanning a page structure	8
C_i	Page isolation: $C_a + W_i$ where W_i is a constant representing locking overhead	28
C_{mf}	Migrate page failure: $C_a \times 2$	16

Table 2.4: Cost of each activity in the Linux kernel. We take W_i to be 20. However, the cost of compaction is not heavily dependent on its exact value.

	At rest		Under stress	
	Linux	**Illuminator**	**Linux**	**Illuminator**
Min	67%	68%	3%	12%
Max	72%	72%	7%	28%
Avg	69%	69%	5%	17%

Table 2.5: Huge page allocation success rate for `stress-highalloc` which tries to allocate 90% of memory as huge pages.

directly impacts the cache efficiency. Page migration also requires TLB shootdowns as the kernel is generally unaware of whether a page is cached in a TLB (on x86). The cost of TLB shootdowns does not scale well and is known to be a major source of performance overhead in multi-core systems [47, 120]. Hence, minimizing $Cost_{mc}$ is essential for multiple reasons.

2.6.3 Huge page allocations with `stress-highalloc`

`stress-highalloc` [13] is a standard benchmark used by the Linux kernel community developers to quantitatively measure the impact of fragmentation mitigation techniques. The benchmark stresses the memory allocators by running multiple kernel compilation jobs in parallel before requesting 90% of total system memory as huge pages. We run `stress-highalloc` with 3GB memory which is also the recommended size for this benchmark.

Table 2.5 shows the success rate of huge page allocations as the average of five runs of `stress-highalloc`. Illuminator behaves similar to Linux when the system is at rest (not fragmented) but outperforms Linux under stress (when memory is heavily fragmented with unmovable pages) due to its better management of unmovable pages. Compared to Linux, Illuminator allocates 3.4× more huge pages and reduces $Cost_{mc}$ by more than 80%.

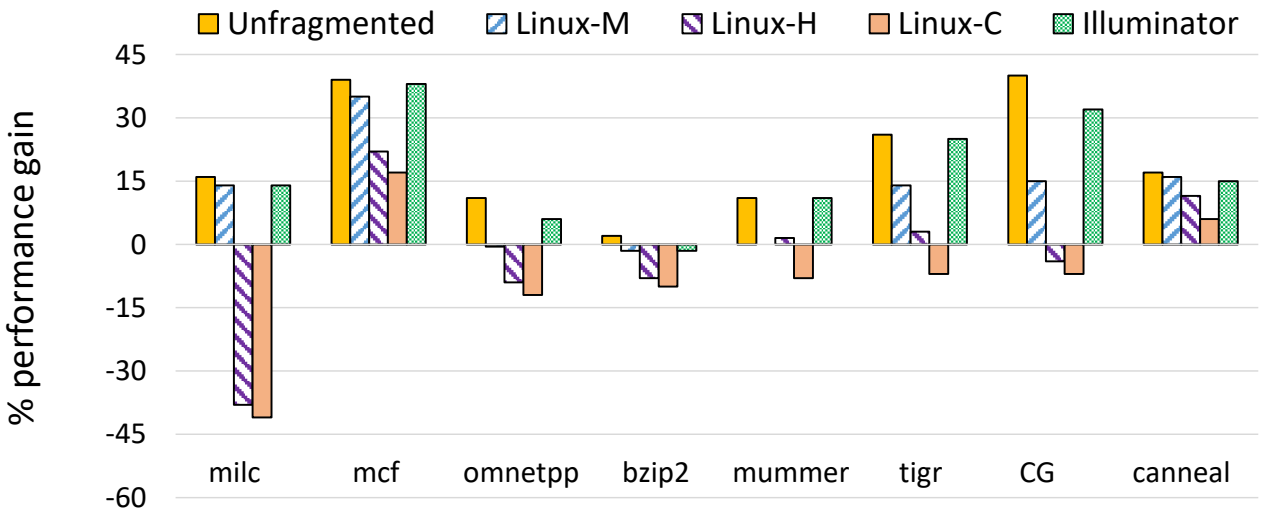


Figure 2.10: Performance relative to baseline pages at 0.25 (Linux-M), 0.5 (Linux-H) and 0.75 (Linux-C) unmovability indices. Illuminator's performance is presented once which is valid for all fragmentation indices considered. Notice that the performance in Linux degrades as fragmentation increases resulting in worse than the performance of baseline pages at 0.75 unmovability index for most applications.

2.6.4 Performance results on bare-metal

The main objective of Illuminator is to mitigate fragmentation in long-running systems. Therefore, assessing the performance benefits of Illuminator requires physical memory to get fragmented. To the best of our knowledge, `stress-highalloc` is the only reliable benchmark available for fragmenting memory with unmovable pages. However, this benchmark is reliable for only up to 3GB physical memory which does not provide a realistic context on modern large memory systems. Hence, we use our synthetic benchmark tool as discussed in Section 2.3 to reliably fragment the physical memory prior to executing our workloads. The benchmark tool repeatedly allocates movable and unmovable page frames leading to many memory fallbacks across movable and unmovable pools of memory. Our tool can be configured to fragment physical memory up to a pre-determined threshold in Linux, defined in terms of UI.

To understand the effect of fragmentation in various stages, we measure performance in three different states of fragmentation i.e., *moderate, high* and *critical*, corresponding to the unmovability index (UI) of 0.25, 0.5 and 0.75, respectively. We refer to Linux at moderate, high and critical fragmentation levels as Linux-M, Linux-H and Linux-C.

We find that Illuminator's performance is consistent despite varying fragmentation because it limits the number of hybrid pageblocks to less than 10% of Linux. Hence, less than 8%

of total pageblocks are hybrid in Illuminator even when Linux has 75% hybrid pageblocks. Hence, we present Illuminator results only once as it is valid across all three fragmentation levels considered. To compare the performance of Linux and Illuminator against the best possible performance attainable with huge pages, we also evaluate workloads in the absence of fragmentation. For this, we record each application's performance on a freshly booted system. This ensures that no huge page allocation request fails for the entire duration of each workload.

2.6.4.1 Overall performance improvement

Figure 2.10 shows performance relative to baseline pages. In Linux, application performance degrades as fragmentation increases. Notice that *most applications perform even worse than baseline pages in Linux-H and Linux-C*. The worst case is seen for `milc` which suffers 38% and 41% performance loss in Linux-H and Linux-C. Even in cases where Linux improves performance, the benefits are significantly lesser than the unfragmented case. For example, Linux-H improves the performance of `mcf` by 22% as compared to 39% of the unfragmented case.

Illuminator outperforms Linux and achieves performance comparable to the unfragmented system for 5 out of 8 applications while others are within 2–12% of the unfragmented system. Some performance loss compared to the best case is expected for multiple reasons: (1) Illuminator eliminates only LIU migration; the overhead of normal migration also affects performance, and (2) under heavy fragmentation, many huge page allocation requests fail, leading to baseline 4KB page allocations for the corresponding application virtual memory regions. Though such memory regions are promoted to huge pages by the kernel in the background, this operation takes time. Therefore, until promotion, applications continue to execute with baseline pages, incurring TLB miss overheads as compared to the unfragmented case where all huge page allocations succeed at the time of page fault.

Application	**Linux-M**	**Linux-H**	**Linux-C**	**Illuminator**
milc	17621/34	4657/209	**65/0**	**17594/45**
mcf	121/6	81/4	17/0	135/15
omnetpp	20/90	15/20	17/17	22/101
bzip2	684/12	190/9	110/0	620/55
mummer	112/0	57/0	11/0	483/9
tigr	88/106	74/72	**11/9**	**228/556**
CG	1349/151	890/134	**793/52**	**3405/736**
canneal	92/15	68/11	31/0	92/379

Table 2.6: Number of huge pages allocated/promoted. Allocation happens in the page fault handler while promotion is done by the `khugepaged` kernel thread in background.

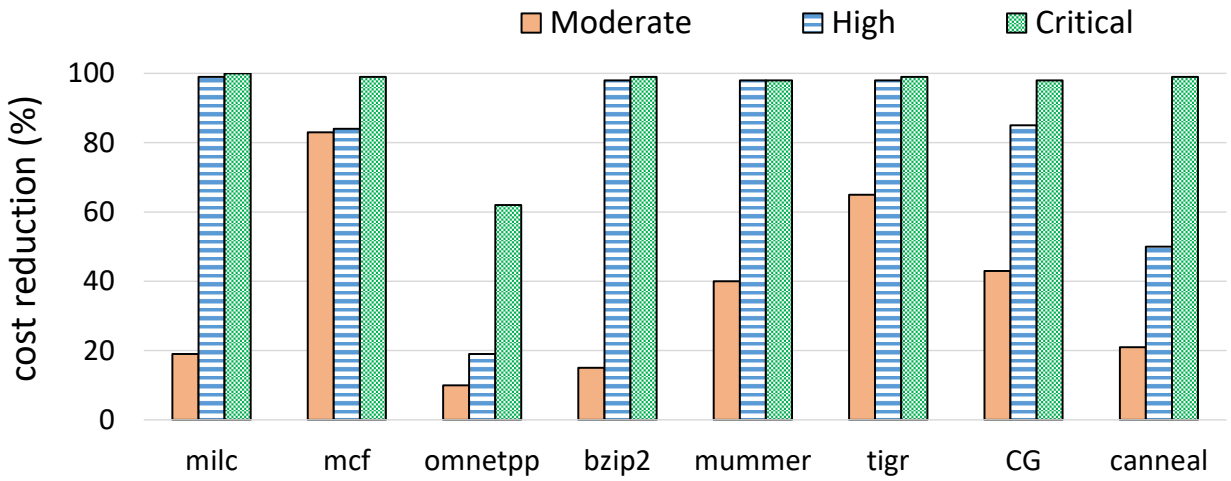


Figure 2.11: Reduction in the cost of compaction with Illuminator at different fragmentation levels.

Despite these factors, Illuminator provides significantly better performance than Linux in all fragmented situations. For example, it improves the performance of `milc` by 55%, `mcf` by 21%, `omnetpp` by 18%, `mummer` by 19%, `tigr` by 32%, `CG` by 38% and `canneal` by 8% compared to Linux-C. The average performance improvement is also significant i.e., 5.5%, 19.5% and 25% compared to Linux-M (UI=0.25), Linux-H (UI=0.5) and Linux-C (UI=0.75).

Next we discuss how Illuminator's better performance is correlated to the number of huge page allocations and the cost of compaction incurred in allocating huge pages.

Huge page allocations: Table 2.6 shows that Linux's ability to allocate huge pages is impacted by the unmovability index (UI). For example, Linux-M allocates about 17K huge pages to `milc` which drops to 4.6K in Linux-H and further to only 65 in Linux-C. Illuminator allocates about 17.5K huge pages to `milc`.

Similarly, `khugepaged`'s ability to promote huge pages in the background also gets impacted by fragmentation in Linux. Notice that Linux-C is unable to promote any huge page for 5 out of 8 workloads. The best case for huge page promotions is observed with `tigr` and `CG` where Illuminator promotes 556 and 736 huge pages compared to 9 and 52 in Linux-C.

Compaction overhead: Illuminator significantly reduces the cost of compaction by eliminating LIU migration (Figure 2.11); the reduction is 10–83% compared to Linux-M (UI=0.25), 19–99% compared to Linux-H (UI=0.5)%, and 62–99% compared to Linux-C (UI=0.75).

Efficient compaction in turn reduces CPU utilization and lowers the amount of time an

	Time spent in kernel mode execution (seconds)	% of total execution time spent in kernel mode
Linux-M	11	2%
Linux-H	255	28%
Linux-C	343	37%
Illuminator	11	2%

Table 2.7: Kernel mode execution time (in seconds) and the percentage of total time spent in the kernel mode for `milc`.

application spends in the kernel mode of execution (i.e., system time). While we observed a significant reduction in the system time of all applications, we report it only for `milc` (see Table 2.7) as the system time was a small fraction (less than 5%) of the overall execution time for other applications. For `milc`, system time is higher because this application stresses the compaction code path due to its repeated memory allocation and freeing patterns. In Linux-M (UI=0.25), `milc` spends only about 2% of its overall execution time in kernel mode which increases to 28% in Linux-H (UI=0.5) and further to 37% in Linux-C (UI=0.75). Illuminator keeps the system time of `milc` within 2% of the overall execution time.

2.6.4.2 Latency and OS jitter

We evaluate the impact of LIU migration on latency with the `MySQL` database server configured with a table of 32 million rows on which a read-only workload was executed with 8 threads. We repeat the experiment 10 times with 30 minutes per iteration and report the maximum latency observed in each iteration (see Figure 2.12).

In Linux, huge page allocation latency is affected by the location of unmovable pages as the time taken to allocate a huge page depends on the number of hybrid pageblocks encountered by the migrate scanner. Such behavior leads to high latency spikes in Linux. For example in Linux-H, the maximum latency of `MySQL` read requests varies from 57ms–4702ms across ten iterations. In Illuminator, the maximum latency across iterations varies from 16ms–156ms, which is much lower than Linux due to the elimination of LIU migration. Notice that eliminating LIU migration alone does not provide any throughput improvement. When fragmentation via pollution is handled along with LIU migration, we observed 14% higher throughput for `MySQL` due to a higher number of huge page allocations.

2.6.4.3 Performance isolation

As discussed earlier, applications like `milc` monopolize memory compaction by generating frequent page faults. Note that compaction is a global process that migrates all movable pages

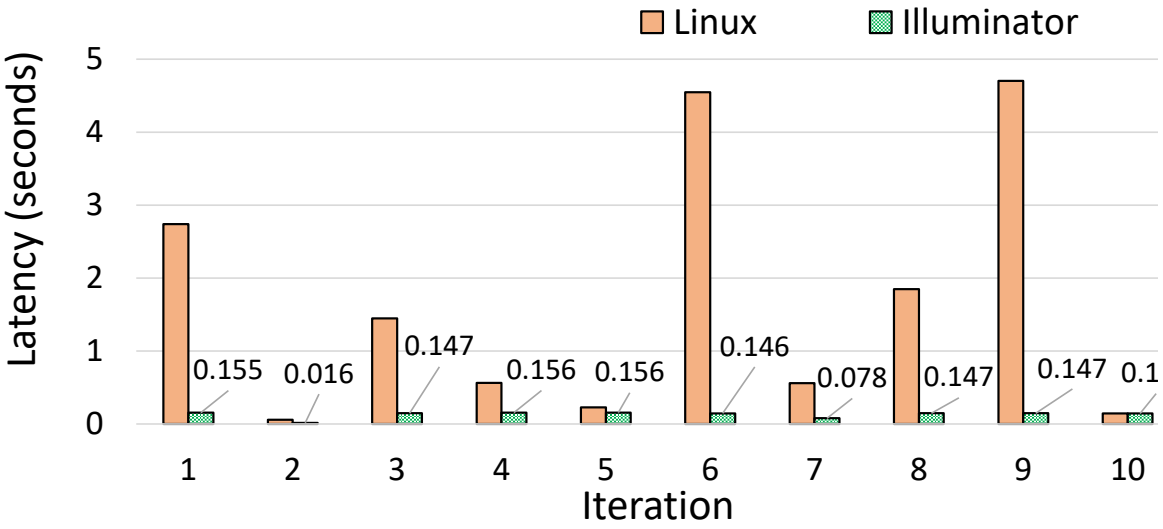


Figure 2.12: Maximum latency for `MySQL` read requests from 10 iterations at high fragmentation level (UI=0.5).

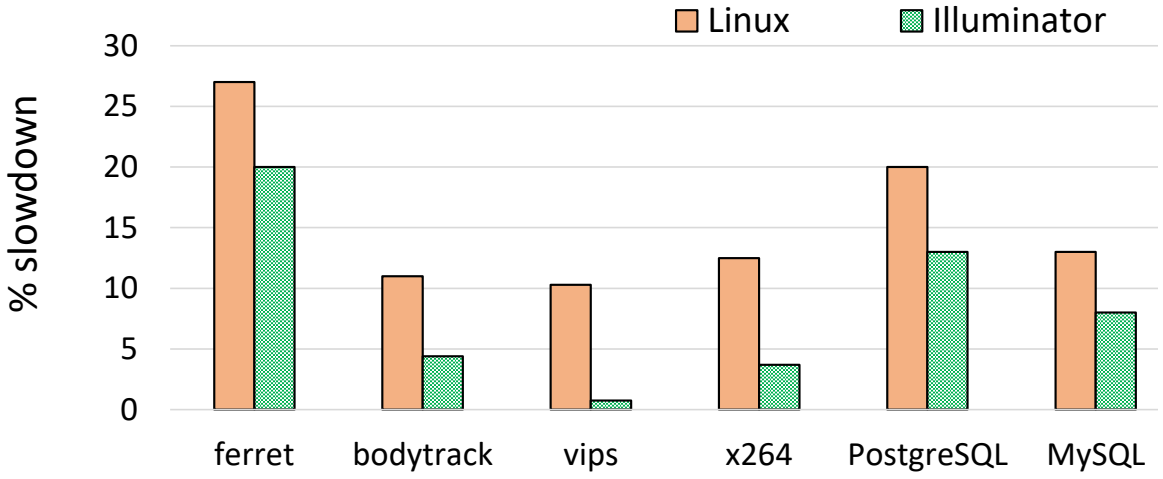


Figure 2.13: Slowdown for applications (lower is better) while running alongside `milc` at high fragmentation.

encountered by the migrate scanner until it either fails, allocates/promotes sufficient huge pages or gets preempted by the scheduler. In a fragmented system, LIU migration can lead Linux into a compaction loop which in turn negatively impacts other workloads running on the same machine.

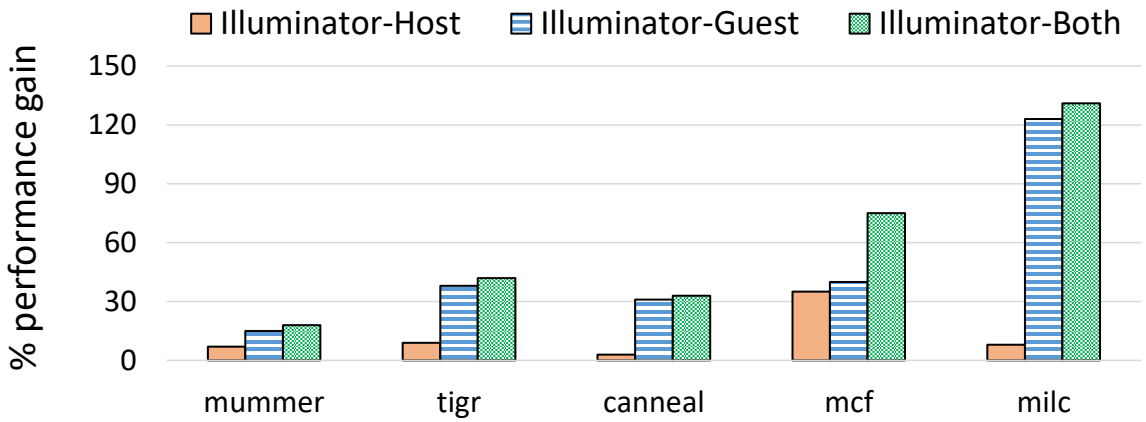


Figure 2.14: Performance improvement over Linux-H (UI=0.5) in a virtualized system when Illuminator is deployed at host, guest and both.

We measure performance isolation by executing a few workloads (one-by-one) alongside `milc` and record the slowdown observed by them in Linux-H and Illuminator. Figure 2.13 shows the slowdown experienced by the workloads when fragmentation is high. Illuminator provides better performance isolation in all cases. For example, `vips` slows down by 10% in Linux-H but experiences negligible performance loss with Illuminator. Performance isolation is considerably better for all other applications as well in Illuminator. We noticed that by avoiding LIU migration, Illuminator reduces the number of TLB shootdowns by 63% (for `ferret`) to 90% (for `vips`).

2.6.5 Performance in virtualized environments

We evaluate a virtualized setup with Ubuntu16.04 running at the host with 24GB memory and kernel virtual machine or KVM as the hypervisor. The guest operating system runs with Ubuntu12.04, 8 cores and 8GB memory. In a virtualized system, application performance depends on the state of memory fragmentation at both the guest and the host layers. For brevity, we report results for five applications at 0.5 unmovability index in both guest and host (see Figure 2.14). Experiments are reported for the three configurations: 1) when Illuminator is applied only at the host, 2) when Illuminator is applied only at the guest, and 3) when Illuminator is applied at both guest and host.

The performance benefits are higher when Illuminator is applied at the guest as compared to when applied at the host, except for `mcf`, for which the improvement is comparable in both

cases. Intuitively, the best performance is observed when Illuminator is operating in both layers. For example, the performance of `mcf` improves by 35% and 41% when Illuminator is deployed at only host or only guest which effectively increases to 75% when it is deployed at both layers. Performance improvement for `mummer`, `tigr` and `canneal` is also substantial i.e., 18%, 42% and 32% in the best case. Similar to the native setup, the best performance improvement is observed for `milc` i.e., 131% as compared to Linux.

In our experiments, guest memory fragmentation results in higher performance loss than host memory fragmentation for two main reasons: 1) our workloads are more sensitive to address translation overhead at the guest, except for `mcf` which is equally sensitive to address translation performance at both layers. As a result, performance loss in Linux due to fragmentation in guest is higher than host memory fragmentation; `mcf` incurs similar performance loss in guest only and host only fragmented scenarios. 2) applications are executed after fragmenting the guest memory. Hence, the overhead of synchronous compaction performed by the host impacts the process that creates fragmentation; applications that execute after memory has been fragmented, encounter only the asynchronous compaction of the host. As discussed previously, the performance implications of asynchronous compaction are modest as compared to the synchronous compaction. In contrast, both synchronous and asynchronous compaction of the guest OS impact applications running inside the guest.

2.7 Discussion

Unmovable pages are found in most general-purpose operating systems [96, 9]. For example, the outermost page tables and DMA pages (without IOMMU support) are inherently unmovable [124]. For large memory workloads, page tables can occupy multiple GBs of memory [164] which highlights the importance of efficiently handling unmovable pages. However, different operating systems can have different unmovability constraints. Operating system designs that provide support for movable kernel pages (e.g., Windows, Solaris) are less likely to suffer from the issues discussed in this chapter. However, not directly mapping memory in the kernel address space leads to a complex design and performance issues. Hence OSs with movable kernel pages also prefer directly mapping at least some part of the kernel virtual address space in physical memory [162]. For example, FreeBSD reserves a special region in the kernel address space for directly mapping the objects allocated by its slab-like zone allocator. The high-level design of its memory manager is also similar to Linux [135, 140]; many of its internal data structures are wired or unmovable [139], and allocations are handled with the combination of a zone allocator and a buddy allocator. Hence, an interesting future work is to port Illuminator to FreeBSD. However, performance implications depend on how much unmovable memory is

present and how the kernel defragments memory. A detailed cross OS study on this topic is a potential future work.

In addition, many prior techniques also demand memory contiguity for improving performance or optimizing energy consumption [188, 55, 91]. Understandably, such techniques are also vulnerable to fragmentation created by unmovable pages. Illuminator can be used to make them more robust in long-running systems.

2.8 Summary

We address the issue of performance regressions caused by huge pages and show how unmovable pages can become a major hindrance to fragmentation mitigation. We propose Illuminator – a simple memory management framework guided by a precise understanding of external memory fragmentation – to make huge page allocations feasible as well as cost-effective in long-running systems. With detailed evaluation, we show that Illuminator provides good performance in both native and virtualized systems. While this chapter has discussed Illuminator only in the context of huge pages, the proposed techniques are generic and can be helpful in mitigating other kernel level fragmentation issues. The source code of Illuminator is available at https://github.com/apanwariisc/Illuminator.

Chapter 3

Fine-grained Huge Page Management

In Chapter 2, we discussed the effect of fragmentation on huge pages, and proposed techniques to make it feasible to allocate huge pages in long-running systems. However, external memory fragmentation is just one of many huge page management challenges for operating systems.

In this chapter, we dive into the details of policy-level OS challenges for huge pages. First, we expose some interesting pitfalls in current huge page management policies and then discuss our proposed system called HawkEye to address the associated challenges.

3.1 Introduction

Modern architectures employ large multi-level TLBs to minimize TLB misses, and further use page-walk caches to avoid accessing physical memory for address translation. These hardware structures support multiple page sizes [121, 149, 60, 76]. Therefore, modern systems require careful OS design to determine suitable page sizes for different workloads [93, 121, 139, 171]. The problem becomes more severe with virtual machines where two layers of address translation cause additional address translation overheads [58, 91, 151].

Despite robust hardware support available across architectures [26], huge pages have provided unsatisfactory performance on important applications [2, 7, 8, 12, 17, 15, 16]. In Chapter 2, we discussed the effect of external fragmentation on various performance anomalies with huge pages. In this chapter, we show that these performance issues also appear due to some other inadequate OS-based huge page management algorithms [1, 3, 4].

OS-based huge page management algorithms need to balance complex trade-offs between address translation overheads (aka MMU overheads), memory bloat due to internal fragmentation in huge pages, page fault latency, fairness and the overheads of the algorithm itself. In this chapter, we discuss some subtleties related to huge pages and expose important weaknesses

of current approaches, and propose a new set of algorithms to address them. We begin with a brief overview of the three representative state-of-the-art systems: Linux, FreeBSD, and a recent research paper by Kwon et al., i.e., Ingens [121].

Linux: Linux's transparent huge page (THP) employs huge pages through two mechanisms: (1) either it allocates a huge page at the time of page fault if contiguous memory is available, or (2) it promotes base pages to huge pages, by optionally compacting memory [69], in a background kernel thread called `khugepaged`. Linux triggers background promotion when external memory fragmentation is high and huge pages are difficult to allocate at the time of page fault. While promoting huge pages, `khugepaged` selects processes in first-come-first-serve (FCFS) order and promotes all huge pages in a process before selecting the next process. For security, a page is zeroed synchronously (except for copy-on-write pages) before getting mapped into the user process' page table.

FreeBSD: FreeBSD supports multiple huge page sizes [139]. Unlike Linux, FreeBSD *reserves* a contiguous region of physical memory in the page fault handler but defers the promotion of baseline pages until all base pages in a huge page sized region are allocated by the application. If the reserved memory region is only partially mapped, its unused pages are returned back to the page allocator when memory pressure increases. This way, FreeBSD manages memory contiguity more efficiently than Linux, at the potential expense of a higher number of page faults and higher MMU overheads due to multiple TLB entries, one per baseline page.

Ingens: In the Ingens paper [121], the authors point out pitfalls in the huge page management policies of both Linux and FreeBSD and present a policy that is better at handling the associated trade-offs. In summary: (1) Ingens uses an adaptive strategy to balance address translation overhead and memory bloat: it uses a conservative utilization-threshold based huge page allocation to prevent memory bloat under high memory pressure but relaxes the threshold to allocate huge pages aggressively under no memory pressure, to try and achieve the best of both worlds. (2) To avoid high page fault latency associated with synchronous page-zeroing, Ingens avoids allocating huge pages in the page fault handler; instead it defers huge page promotion to an asynchronous background thread. (3) To maintain fairness across multiple processes, Ingens treats memory contiguity as a resource and employs a share-based policy to allocate huge pages fairly. Using memory access patterns discovered at runtime, Ingens adjusts the per-process share value to guide the allocation of huge pages.

The Ingens paper highlights that current OSs deal with huge page management issues through "spot fixes", and motivates the need for a principled approach. While Ingens proposes a more sophisticated strategy than previous work, we show that its static configuration based and heuristic-driven approaches are suboptimal: (1) Ingens relies on user input to han-

dle the trade-offs involved in huge pages (e.g., whether to minimize memory footprint or TLB misses). In general, it is hard for users to identify suitable configuration parameters. (2) Ingens allocates huge pages using heuristics and lacks adequate knowledge of the actual MMU overheads experienced by the application. This limits the profitability of huge pages allocated by Ingens. Overall, we find that several aspects of an OS-based huge page management system can benefit from a dynamic data-driven approach.

This chapter of the thesis presents HawkEye, an automated OS-level solution for huge page management. HawkEye proposes a set of simple-yet-effective algorithms to: (1) balance the trade-offs between memory bloat, address translation overheads, and page fault latency, (2) allocate huge pages to applications with highest expected performance improvement due to the allocation, and (3) improve memory sharing behaviour in virtualized systems. We also show that the actual address-translation overheads, as measured through performance counters, can be sometimes quite different from the expected/estimated overheads. To demonstrate this, we also evaluate a variant of HawkEye that relies on hardware performance counters for making huge page allocation decisions, and compare results. Our evaluation involves workloads with a diverse set of requirements vis-a-vis huge page management, and demonstrates that HawkEye can achieve considerable performance improvement over existing systems while adding negligible overhead ($\approx$ 3.4% single-core overhead in the worst-case).

3.2 Motivation

In this section, we discuss different tradeoffs involved in OS-level huge page management and how current solutions handle them. We also provide a glimpse of the results achieved through HawkEye which is discussed in detail in Section 3.3.

3.2.1 Address translation overhead vs. memory bloat

One of the most important tradeoffs associated with huge pages is between MMU overheads and memory footprint of an application [121, 128, 152]. Note that a huge (e.g., 2MB) page requires a large contiguous physical memory block. However, the entire huge page may not be accessed by the application e.g., due to sparse access patterns or internal fragmentation in the heap. Therefore, huge pages can result in wasted physical memory which is typically referred to as memory bloat. In data-center applications, huge pages have been observed to increase memory footprint by as much as 78% [128].

While Linux's synchronous huge page allocation is aimed at minimizing MMU overheads, it can often lead to memory bloat when an application uses only a fraction of the allocated huge page. FreeBSD promotes only after all base pages in a huge page region are allocated.

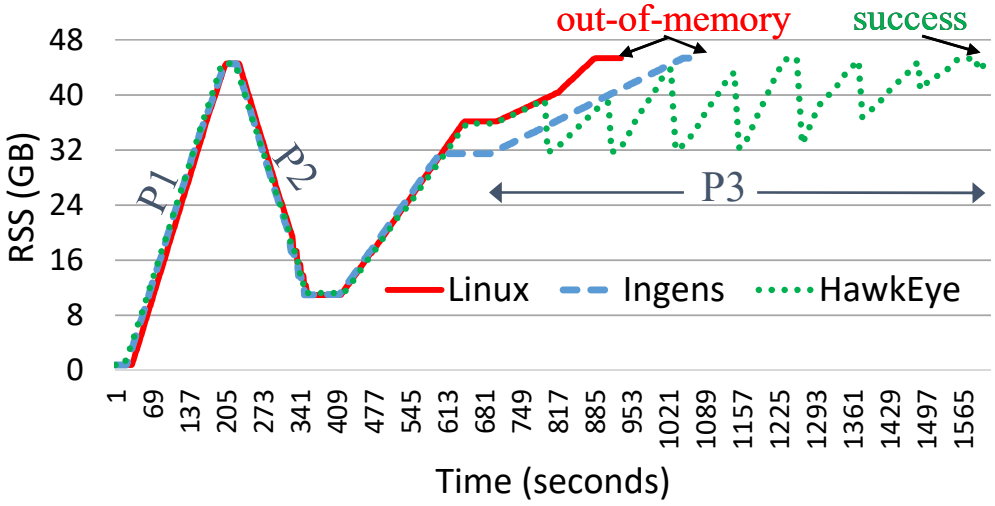


Figure 3.1: Resident Set Size (RSS) of `Redis` server across 3 phases: P1 (insert), P2 (delete) and P3 (insert).

Therefore, FreeBSD's conservative huge page allocation approach tackles memory bloat more efficiently but sacrifices performance by delaying the mapping of huge pages.

Ingens' strategy is adaptive, in that it promotes huge pages aggressively to minimize MMU overheads when there is enough free physical memory is available in the system. Ingens estimates the availability of free physical memory using the Free Memory Fragmentation Index (FMFI is discussed in detail in Section 2.2). A low value of FMFI indicates a low memory pressure situation. Therefore, if FMFI < 0.5 (low fragmentation), Ingens behaves like Linux and promotes allocated pages to huge pages at the first available opportunity. When FMFI > 0.5 (high fragmentation), Ingens uses a conservative utilization-based strategy (i.e., promotes only after a certain fraction of base pages, say 90%, are allocated) to mitigate memory bloat. While Ingens appears to capture the best of both Linux and FreeBSD, we show that this adaptive policy is far from a complete solution because *memory bloat generated in the aggressive phase remains unrecovered*. This property is also true for Linux and we demonstrate this problem through a simple experiment with the `Redis` key-value store [67] running on a 48GB memory system (see Figure 3.1).

We execute a client to generate memory bloat and measure the interaction of different huge page policies with `Redis`. For exposition, we divide our workload into three phases: (P1) The client inserts 11 million key-value pairs of size (10B, 4KB) for an in-memory dataset of 45GB. (P2) The client deletes 80% randomly selected keys leaving `Redis` with a sparsely populated address space. (P3) After some time gap, the client tries to insert 17K (10B, 2MB) key-value pairs so that the dataset again reaches 45GB. While this workload is used for clear exposition

of the problem, it resembles realistic workload patterns that involve a mix of allocation and deallocation, and leave the address space fragmented. For example, many of Google's data-centers workloads exhibit this type of behavior [128]. An ideal system should recover elegantly from memory bloat to avoid disruptions under high memory pressure.

In phase P2, when randomly selected keys are deleted, `Redis` releases the corresponding pages back to the OS through `madvise` system call [6]. In response, the kernel breaks the corresponding huge page mappings, reducing the resident-set size (RSS) of `Redis` to around 11GB. At this point, the kernel's `khugepaged` thread promotes the regions containing the deallocated pages back to huge pages. As more allocations happen in phase P3, Ingens promotes huge pages aggressively until the RSS reaches 32GB (or fragmentation is low); after that, Ingens employs conservative promotion to avoid any further bloat. This is evident in Figure 3.1 where the rate of RSS growth decreases, compared to Linux. However both Linux and Ingens reach memory limit (out-of-memory OOM exception is thrown) at significantly lower memory utilization. While Linux generates total memory bloat of 28GB (i.e., only 20GB useful data), Ingens generates 20GB memory bloat (i.e., 28GB useful data). Note that although Ingens tries to prevent bloat at high memory utilization, it is unable to recover from bloat that gets generated in the low memory utilization phase.

We note that it is possible to avoid memory bloat in Ingens by configuring it to use only conservative promotion. However, as also explained in the Ingens paper, this strategy risks losing performance due to high MMU overheads in low memory pressure situations. An ideal strategy should promote huge pages aggressively but should also be able to recover from bloat in an elegant fashion. Figure 3.1 shows HawkEye's behaviour for this workload, which is able to effectively recover from memory bloat, even with an aggressive promotion strategy.

3.2.2 Page fault latency vs. number of page faults

The OS often needs to clear (zero) a page before mapping it into the process address space to prevent insecure information flow across applications. Clearing a page is significantly more expensive for huge pages: in Linux on our experimental system, clearing a base page takes about 25% of total page fault time which increases to 97% for huge pages! High page fault latency often leads to high user-perceived latencies, jeopardizing performance for interactive applications. Ingens avoids this problem by allocating only base pages in the page fault handler and relegating the promotion task to an asynchronous thread `khugepaged`. To account for temporal locality, it prioritizes the promotion of recently faulted regions over older allocations.

While Ingens reduces page allocation latency, it nullifies an important advantage of huge pages, namely fewer page faults for access patterns exhibiting high spatial locality [33]. Several

Event	sync page-zeroing		async promotion	no page-zeroing	
	Linux 4KB	Linux 2MB	Ingens 90%	Linux 4KB	Linux 2MB
# Page faults	26.2M	51.5K	26.2M	26.2M	51.5K
Total page fault time (secs)	92.6	23.9	92.8	69.5	0.7
Avg page fault time (μs)	3.5	465	3.5	2.65	13
System time (secs)	102	24	104	79	1.3
Total time (secs)	106	24.9	116	83	4.4

Table 3.1: Page faults, allocation latency and performance for a microbenchmark with ≈100GB memory allocation.

applications that allocate and initialize a large memory region sequentially exhibit this type of access pattern and their performance degrades due to higher number of page faults if only base pages are allocated. We demonstrate the severity of this problem through a custom micro-benchmark that allocates a 10GB buffer, touching one byte in every base page and later frees the buffer. Table 3.1 shows the cumulative performance summary for 10 runs of this micro-benchmark.

Linux with THP support (Linux-2MB with sync page-zeroing in Table 3.1) reduces the number of page faults by more than 500× over Linux without THP support (Linux-4KB) and leads to more than 4× performance improvement for this workload, despite being 133× worse on average page fault latency (465μs vs. 3.5μs). Ingens considerably reduces latency compared to Linux-2MB but does not reduce the number of page faults for this workload because asynchronous promotion is activated only after 90% base pages are allocated from a region; copying these pages to a contiguous memory block takes more time than the time taken by this workload to generate the remaining 10% page faults in the same region. Consequently, the overall performance of this workload degrades in Ingens due to excessive page faults. If it were possible to allocate pages without having to zero them at page fault time, we could achieve *both* low page fault latency and fewer page faults resulting in higher overall performance over all existing approaches (last two columns in Table 3.1). HawkEye implements rate-limited asynchronous page pre-zeroing (Subsection 3.3.1) to achieve this in the common case.

3.2.3 Huge page allocation across multiple processes

Another policy challenge that appears for a huge page management system is about how to allocate huge pages when the demand for huge pages exceed the supply of contiguous physical memory? As discussed in Chapter 2, physical memory fragmentation is a common occurrence in long-running systems. Under such situations, the OS must allocate huge pages among multiple

processes *fairly*. Ingens authors tackled this problem by defining fairness through a *proportional huge page promotion* wherein they consider "memory contiguity as a resource" and try and be equitable in distributing it (through huge pages) among applications.

Ingens allocates huge pages based on per-process huge page share priority (say, S). To account for workload-specific requirement, Ingens starts each application with the same value of S but adjusts it at runtime by penalizing applications that have been allocated huge pages but are not accessing them frequently (i.e., have idle huge pages). It estimates idleness of a page through the access-bit present in the page table entries which is set by the hardware and periodically cleared by the OS. Ingens employs an *idleness penalty factor* whereby an application is penalized for its idle, or *cold*, huge pages during the computation of S. The idleness penalty factor is a tunable parameter in Ingens.

We find that Ingens' fairness policy has an important weakness — two processes may have similar huge page requirements but one of them (say P1) may have significantly higher TLB pressure than the other (say P2). This can happen, for example, if P1's accesses are spread across many base pages within its huge page regions, while P2's accesses are concentrated in one (or a few) base pages within its huge page regions. In this scenario, promotion of huge pages in P1 is more desirable than P2 but Ingens would treat them equally.

Table 3.2 provides some empirical evidence showing that different applications usually behave quite differently vis-a-vis huge pages. For example, less than 20% of the total applications in popular benchmark suites experience noticeable performance improvement ($> 3\%$) with huge pages.

We posit that instead of considering "memory contiguity as a resource" and granting it equally among processes, it is more effective to consider "MMU overhead as a system overhead" and try and ensure that it is distributed equally across processes. A fair algorithm should attempt to equalize MMU overheads across all applications, e.g., if two processes P1 and P2 experience 30% and 10% MMU overheads respectively, then huge pages should be allocated to P1 until its overhead also reaches 10%. In doing so, it should be acceptable if P1 has to be allocated more huge pages than P2. Such a policy would additionally yield the best overall system performance by helping the most afflicted processes first.

Further, in Ingens, huge page promotion is triggered in response to a few page-faults in the aggressive (non-fragmented) phase. These huge pages are not necessarily frequently accessed by the application; yet they contribute to a process's allocation quota of huge pages. Under memory pressure, these idle huge pages lower the huge page share S of a process, potentially preventing the OS from allocating more huge pages to it. This would increase MMU overheads if the process had other *hot* (frequently accessed) memory regions that required huge page

Benchmark Suite	Number of applications	
	Total	TLB sensitive applications
SPEC CPU2006_int	12	4 (`mcf, astar, omnetpp, xalancbmk`)
SPEC CPU2006_fp	19	3 (`zeusmp, GemsFDTD, cactusADM`)
PARSEC	13	2 (`canneal, dedup`)
SPLASH-2	10	0
Biobench	9	2 (`tigr, mummer`)
NPB	9	2 (`cg, bt`)
CloudSuite	7	2 (`graph-analytics, data-analytics`)
Total	79	15

Table 3.2: Number of TLB sensitive applications in popular benchmark suites. We consider an application to be TLB sensitive if its address translation overhead is more than 3%.

promotion. This behaviour where previously-allocated cold huge pages can prevent an application from being allocated huge pages for its frequently accessed regions seems sub-optimal and avoidable.

Finally, within a process, Linux and Ingens promote huge pages through a *sequential* scan from lower to higher virtual addresses (VAs). This approach is unfair to processes whose hot regions lie in the higher VAs. Because different applications would usually contain hot regions in different parts of their VA spaces (see Figure 3.6a and Figure 3.7a), this scheme is likely to cause unfairness in practice.

3.2.4 How to capture address translation overheads?

It is common to estimate MMU overheads based on working-set size (WSS) [129]: in expectation, bigger WSS should entail higher MMU and performance overheads. We find that this is often not true for modern hardware where access patterns play an important role in determining MMU overheads, e.g., sequential access patterns allow prefetching to hide TLB miss latencies. Further, different TLB miss requests can experience high latency variations: a translation may be present anywhere in the multi-level page walk caches, multi-level regular data caches or in main memory. For these reasons, WSS is often not a good indicator of MMU overheads. Table 3.3 demonstrates this with the NAS Parallel Benchmark Suite [49]: a workload with large WSS (e.g., `mg.D`) can have low MMU overheads compared to one with a smaller WSS (e.g., `cg.D`).

It is not clear to us how an OS can reliably capture MMU overheads: these are dependent on complex interactions between applications and the underlying hardware architecture, and we show that the actual address translation overheads of a workload can be quite different

Workload	RSS	WSS	% TLB-misses (native-4KB)	% cycles		speedup	
				4KB	2MB	native	virtual
bt.D	10GB	7-10 GB	0.45	6.4	1.31	1.05	1.15
sp.D	12GB	8-12 GB	0.48	4.7	0.25	1.01	1.06
lu.D	8GB	8 GB	0.06	3.3	0.18	1.0	1.01
mg.D	26GB	24 GB	0.03	1.04	0.04	1.01	1.11
cg.D	16GB	7-8 GB	28.57	39	0.02	1.62	2.7
ft.D	78 GB	7-35 GB	0.21	3.9	2.14	1.01	1.04
ua.D	9.6 GB	5-7 GB	0.01	0.8	0.03	1.01	1.03

Table 3.3: Memory characteristics i.e., resident set size (RSS), working set size (WSS), address translation overheads and speedup huge pages provide over base pages for NPB workloads. % cycles denote the fraction of total CPU cycles spend in address translation.

	Performance Counter
C1	DTLB_LOAD_MISSES.WALK_DURATION
C2	DTLB_STORE_MISSES.WALK_DURATION
C3	CPU_CLK_UNHALTED
MMU Overhead = ((C1 + C2) * 100) / C3	

Table 3.4: Methodology used to measure MMU Overhead [114].

from what can be estimated through its memory access pattern (e.g., working-set size). Hence, we propose directly measuring TLB miss overheads through hardware performance counters when available (see Table 3.4 for methodology). This approach enables a more efficient solution at the cost of portability as the required performance counters may not be available on all platforms (e.g., most hypervisors have not yet virtualized TLB-related performance counters). To overcome this challenge, we present two variants of our algorithm/implementation in HawkEye/Linux, one where the MMU overheads are *measured* through hardware performance counters (HawkEye-PMU), and another where MMU overheads are *estimated* through the memory access pattern (HawkEye-G). We compare both approaches in Section 3.4.

3.3 Design and Implementation

Figure 3.2 shows our high-level design objectives. Our solution is based on four primary observations: (1) high page fault latency for huge pages can be avoided by asynchronously pre-zeroing free pages; (2) memory bloat can be tackled by identifying and de-duplicating zero-filled baseline pages present within allocated huge pages; (3) promotion decisions should be based on finer-grained access tracking of huge page sized regions, and should include recency, frequency, and *access-coverage* (i.e., how many baseline pages are accessed inside a huge page) measurements;

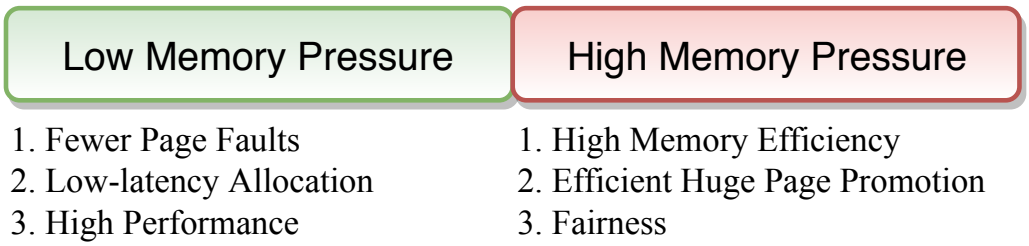


Figure 3.2: Our design objectives in HawkEye.

and (4) fairness should be based on estimation of MMU overheads.

3.3.1 Asynchronous page pre-zeroing

We propose that page zeroing of available free pages should be performed asynchronously in a rate-limited background kernel thread to eliminate high latency allocation. We call this scheme *async pre-zeroing*. Async pre-zeroing is a rather old idea and had been discussed extensively among kernel developers in early 2000s [18, 19, 24, 85]*. Linux developers opined that async pre-zeroing is not an overall performance win for two main reasons. We think that it is time to revisit these opinions.

First, the async pre-zeroing thread might interfere with primary workloads by polluting hardware data caches [18]. In particular, async pre-zeroing suffers from the "double cache miss" problem because it causes the same datum to be accessed twice with a large re-use distance: first for pre-zeroing, and then for the actual access by the application. These extra cache misses are expected to degrade overall performance in general. However, these problems are partially solvable on modern hardware that support memory writes with non-temporal hints: non-temporal hints instruct the hardware to bypass caches during memory load/store instructions [87]. We find that using non-temporal hints during pre-zeroing significantly reduces both cache contention and the double cache miss problem.

Second, there was no consensus or empirical evidence to demonstrate the benefits of page pre-zeroing for real workloads [18, 86]. We note that the early discussions on page pre-zeroing were evaluating trade-offs with baseline 4KB pages. Our experiments corroborate the kernel developers' observation that despite reducing the page fault overhead by 25%, pre-zeroing does not necessarily enable high performance with 4KB pages. At the same time, we also show that it enables non-negligible performance improvements (e.g., 14× faster VM boot-time) with huge pages, due to much higher reduction (97%) in page fault overheads. Since huge pages (and

*Windows and FreeBSD implement async pre-zeroing in a limited fashion [32, 22]. However, these operating systems do not allocate transparent huge pages eagerly in the page fault handler. Therefore, this idea is more relevant to Linux because it uses synchronous huge page allocations.

huge-huge pages) are supported by most general-purpose processors today [26, 41], pre-zeroing pages is an important optimization that is worth revisiting.

Pre-zeroing offers another advantage for virtualized systems: it increases the number of zero pages in the guest's physical address (GPA) space enabling opportunities for content-based page-sharing at the virtualization host. We evaluate this aspect in Section 3.4.

To implement async pre-zeroing, HawkEye manages free pages in the Linux buddy allocator through two lists: `zero` and `non-zero`. Pages released by applications are first added to the `non-zero` list while the `zero` list is preferentially used for allocation. A rate-limited thread periodically transfers pages from `non-zero` to `zero` lists after zero-filling them using non-temporal writes. Because pre-zeroing involves sequential memory accesses, non-temporal store instructions provide performance similar to regular (caching) store instructions, but without polluting the cache [21]. Finally, we note that for copy-on-write or filesystem-backed memory regions, pre-zeroing may sometimes be unnecessary and wasteful. This problem is avoidable by preferentially allocating pages for these memory regions from the `non-zero` list.

Overall, we believe that async pre-zeroing is a compelling idea for modern workload requirements and modern hardware support. In our evaluation, we provide some early evidence to corroborate this claim with different workloads.

3.3.2 Managing memory bloat vs. address translation performance

To manage the trade-offs between memory bloat and address translation overhead, Linux and FreeBSD use a static design-time policy favoring different optimizations. While Linux eagerly allocates huge pages at the first opportunity to minimize TLB coverage, FreeBSD uses lazy allocation to favor reducing memory footprint. Clearly, none of these approaches are ideal e.g., Linux is suboptimal in terms of memory footprint and FreeBSD is suboptimal in terms of TLB coverage. Ingens improves upon prior approaches but it relies on user inputs and is susceptible to parameter misconfiguration.

We find that it is possible to build a dynamic system that can automatically find a sweet-spot in trade-off space that exists for memory bloat and address translation. A dynamic approach is also desirable because the amount of physical memory that is available at runtime is unknown to users and developers. For example, in a virtualized environment, the amount of memory available depends on the need of co-running workloads. Therefore, an OS should adjust huge page allocations based on the dynamic state of the system. As long as enough free memory is available, it should be utilized to aggressively minimize MMU overheads. However, when physical memory becomes a contended resource, then the OS should reclaim memory bloat from huge pages to avoid out-of-memory failures or disk-based swapping. HawkEye achieves

this objective without relying on user input.

Our approach stems from the insight that most allocations in large-memory workloads are typically "zero-filled page allocations"; the remaining are either filesystem-backed (e.g., through `mmap`) or copy-on-write (COW) pages. However, huge pages in modern scale-out workloads are primarily used for "anonymous" pages that are initially zero-filled by the kernel [114], e.g., Linux supports huge pages only for anonymous memory. This property of typical workloads and Linux allows automatic recovery of bloat under memory pressure.

To state our approach succinctly: we allocate huge pages at the time of first page-fault; but under memory pressure, to recover unused memory, we scan existing huge pages to identify zero-filled baseline pages within them. If the number of zero-filled baseline pages inside a huge page is significant (i.e., beyond a threshold), we break the huge page into its constituent baseline pages and de-duplicate the zero-filled baseline pages to a canonical zero-page through standard COW page management techniques [64]. In this approach, it is possible for applications' in-use zero-pages to also get de-duplicated. While this can result in a marginally higher number of COW page faults in rare situations, this does not compromise correctness.

To trigger recovery from memory bloat, HawkEye uses two watermarks on the amount of allocated memory in the system: `high` and `low`. When the amount of allocated memory exceeds `high` (85% in our prototype), a rate-limited `bloat-recovery` thread is activated which executes periodically until the allocated memory falls below `low` (70% in our prototype). At each step, the `bloat-recovery` thread chooses the application whose huge page allocations need to be scanned (and potentially demoted) based on the estimated MMU overheads of that application: the application with the *lowest* estimated MMU overheads is chosen first for scanning. This strategy ensures that the application that least requires huge pages is considered first — this is consistent with our huge page allocation strategy (Subsection 3.2.3).

While scanning a baseline page to verify if it is zero-filled, we stop on encountering the first non-zero byte in it. In practice, the number of bytes that need to be scanned per in-use (not zero-filled) page before a non-zero byte is encountered is very small: we measured this over a total of 56 diverse workloads, and found that the average offset of the first non-zero byte in a 4KB page is only 9.11 (see Figure 3.3). Hence, only ten bytes need to be scanned on average per in-use application page. For bloat pages however, all 4096 bytes need to be scanned. This implies that the overheads of our `bloat-recovery` thread are largely proportional to the number of bloat pages in the system, and *not* to the total size of the allocated memory. This is an important property that allows our method to scale to large memory systems.

We note that our `bloat-recovery` procedure has many similarities with the standard de-duplication kernel threads used for content-based page sharing for virtual machines [182], e.g.,

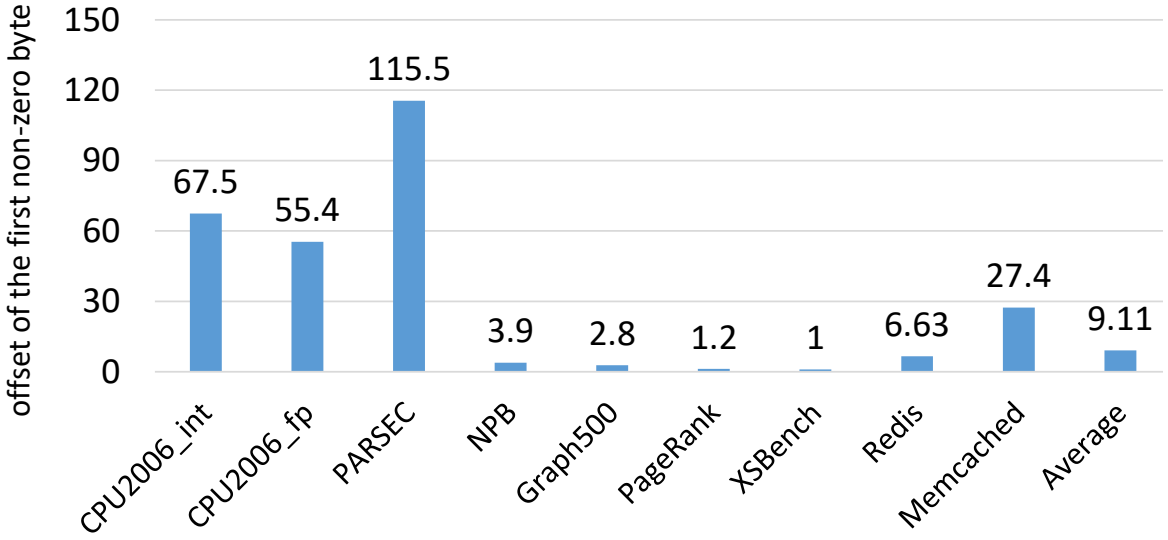


Figure 3.3: Average offset to the first non-zero byte in baseline (4KB) pages. First four bars represent the average of all workloads in the respective benchmark suite.

the kernel same-page merging (`ksm`) thread in Linux [142]. Unfortunately, in current kernels, the huge page management logic (e.g., `khugepaged`) and the content-based page sharing logic (e.g., `ksm`) are unconnected and can often interact in counter-productive ways [102]. Ingens and SmartMD [100] proposed coordinated mechanisms to avoid such conflicts: Ingens demotes only infrequently-accessed huge pages through `ksm` while SmartMD demotes pages based on access-frequency and repetition rate (i.e., the number of shareable pages within a huge page). These techniques are useful for in-use pages and our `bloat-recovery` proposal complements them by identifying unused zero-filled pages, which can execute much faster than the typical same-page merging logic.

3.3.3 Fine-grained huge page promotion

An efficient huge page promotion strategy should try to maximize performance with a minimal number of huge page promotions. Current systems promote pages through a *sequential* scan from lower to higher VAs which is inefficient for applications whose TLB sensitive memory regions are not in lower part of the virtual address space. Our approach makes promotion decisions based on memory access patterns. First we define an important metric used in HawkEye to maximize the performance benefits of huge page promotions.

Access-coverage: it denotes the number of base pages that are accessed from a huge page sized region in a short interval.

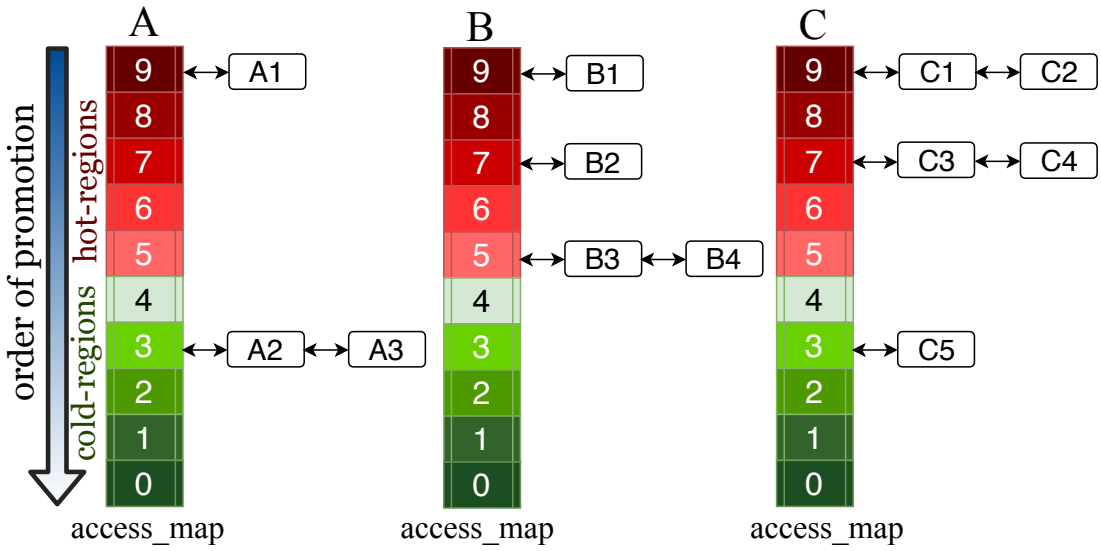


Figure 3.4: A sample representation of `access_map` for three processes A, B and C.

We sample the page table access bits at regular intervals and maintain the exponential moving average (EMA) of access-coverage across different samples. More precisely, we clear the page table access bits and test the number of *set* bits after 1 second to check how many pages are accessed. This process is repeated once every 30 seconds.

The access-coverage of a region potentially indicates its TLB space requirement (number of base page entries required in the TLB), which we use to *estimate* the profitability of promoting it to a huge page. It also captures the relative TLB miss frequency of different huge page sized regions. A region with high access-coverage is likely to exhibit high contention on TLB and hence likely to benefit more from promotion.

HawkEye implements access-coverage based promotion using a per-process data structure, called `access_map`, which is an array of buckets: a bucket contains huge page regions with similar access-coverage. On x86 with 2MB huge pages, the value of access-coverage is in the range of 0-512. In our prototype, we maintain ten buckets in the `access_map` which provides the necessary resolution across access-coverage values at relatively low overheads: regions with access-coverage of 0-49 are placed in bucket 0, regions with access-coverage of $50-99$ are placed in bucket 1, and so on. Figure 3.4 shows an example state of the `access_map` of three processes A, B and C. Regions can move up or down in their respective arrays after every sampling period, depending on their newly computed EMA-based access-coverage. If a region moves up in `access_map`, we add it to the head of its bucket. If a region moves down, we add it to the tail. Within a bucket, pages are promoted from head to tail. This strategy helps in prioritizing recently accessed regions within an index.

HawkEye promotes regions from higher to lower indices in `access_map`. Notice that our approach captures both recency and frequency of accesses: a region that has not been accessed or accessed with low access-coverage in recent sampling periods is likely to shift towards a lower index in `access_map` or towards the tail in its current index. Promotion of cold regions is thus automatically deferred to prioritize recently accessed regions.

We note that HawkEye's `access_map` data structure is somewhat similar to `population_map` [139] and `access_bitvector` [121] used in FreeBSD and Ingens resp.; these data structures are primarily used to capture utilization or page access related metadata at huge page granularity. HawkEye's `access_map` additionally provides the index of hotness of memory regions and enables fine-grained huge page promotion.

3.3.4 Huge page allocation across multiple processes

In our access-coverage based strategy, regions belonging to a process with the highest expected MMU overheads are promoted before others. This is also consistent with our notion of fairness, as pages are promoted first from regions/processes with high expected TLB pressure (and consequently high expected MMU overheads).

Our HawkEye variant that does not rely on hardware performance counters (i.e., HawkEye-G), promotes regions from the non-empty *highest access-coverage index* in all processes. It is possible that multiple processes have non-empty buckets in the (globally) highest non-empty index. In this case, round-robin is used to ensure fairness among such processes. We explain this further with an example. Consider three applications A, B and C with their VA regions arranged in the `access_map` as shown in Figure 3.4. HawkEye-G promotes regions in the following order in this example:

A1,B1,C1,C2,B2,C3,C4,B3,B4,A2,C5,A3

Recall however that MMU overheads may not necessarily be correlated with our access-coverage based estimation, and may depend on other more complex features of the access pattern (see Subsection 3.2.4). To capture this in HawkEye-PMU, we first choose the process with the highest measured MMU overheads, and then choose regions from higher to lower indices in selected process's `access_map`. Among processes with similar highest MMU overheads, we use round-robin to ensure fairness.

3.3.5 Limitations and discussion

We briefly outline a few important aspects related to huge page management that we have not currently handled in HawkEye.

1) Identifying thresholds to detect memory pressure: We measure the extent of memory

pressure with statically configured values for `low` and `high` watermarks (i.e., 70% and 85% of total system memory) while dealing with memory bloat. This approach is inline with Linux's methodology to estimate memory pressure situations. However, any strategy that relies on static thresholds faces the risk of being conservative or overly aggressive when memory pressure consistently fluctuates. An ideal solution should adjust these thresholds dynamically to prevent unintended system behavior. The approach proposed by Guo et. al. [102] in the context of memory deduplication for virtualized environments is relevant in this context.

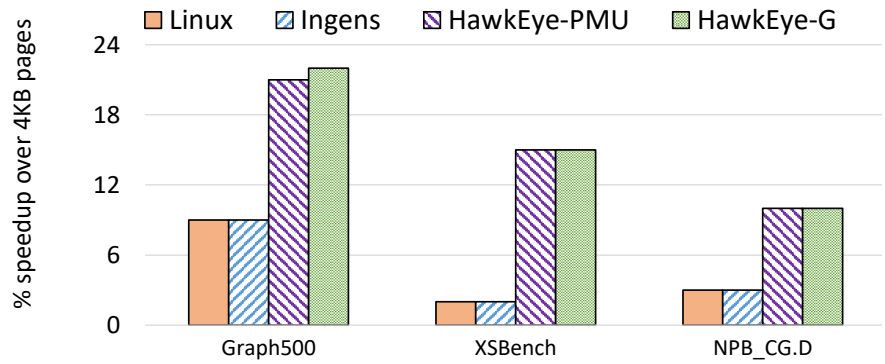
2) Putting a hard limit on huge page allocation: While we believe our approach of allocating huge pages based on MMU overheads optimizes the system as a whole, unbounded huge page allocations to a single process can be thought of as a *starvation* problem for other processes. An adverserial application can also potentially monopolize HawkEye to get more huge pages or prevent other applications from getting a fair share of huge pages. Preliminary investigations show that our approach is reasonable even if the memory footprint of workloads differ by more than an order of magnitude. However, if limiting huge page allocations is still desirable, it seems reasonable to integrate a policy with existing resource limiting/monitoring tools, such as `cgroups` in Linux [31].

3) Other algorithms: We do not discuss some parts of the management algorithms, such as rate limiting `khugepaged` to reduce promotion overheads, demotion based on low utilization to enable better sharing through same-page merging, and techniques for minimizing the overhead of page-table access-bit tracking. Much of this material has been discussed and evaluated extensively in the literature [55, 121, 189], and we have not contributed significantly new approaches in these areas.

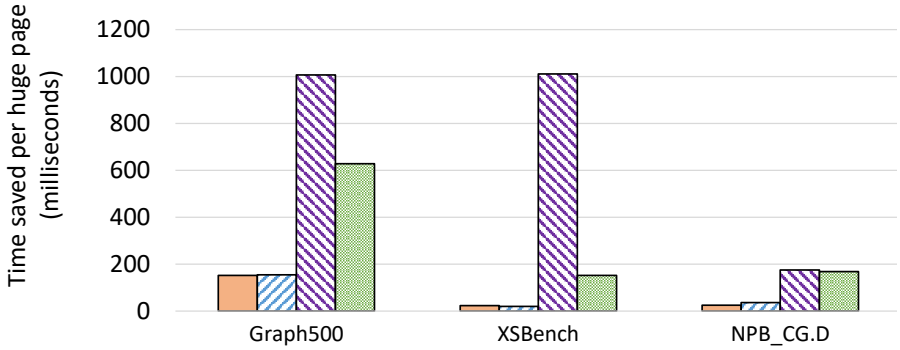
3.4 Evaluation

We now evaluate our algorithms in more detail. Our experimental platform is an Intel Haswell-EP based E5-2690 v3 server system running CentOS v7.4, on which 96GB memory and 48 cores (with hyperthreading enabled) running at 2.3GHz are partitioned on two sockets: we bind each workload to a single socket to avoid NUMA effects. The L1 TLB contains 64 and 8 entries for 4KB and 2MB pages respectively while the L2 TLB contains 1024 entries for both 4KB and 2MB pages. The size of L1, L2 and shared L3 cache is 768KB, 3MB and 30MB resp. A 96GB SSD-backed swap partition is used to evaluate performance in an overcommitted system. We evaluate HawkEye with a diverse set of workloads ranging from HPC, graph algorithms, in-memory databases, genomics and machine learning [12, 176, 49, 56, 62, 43, 105, 177]. To ensure a fair comparison with Ingens, we implemented HawkEye in Linux kernel version 4.3.

We evaluate (a) the improvements due to our fine-grained promotion strategy based on



(a) Performance speedup over 4KB pages



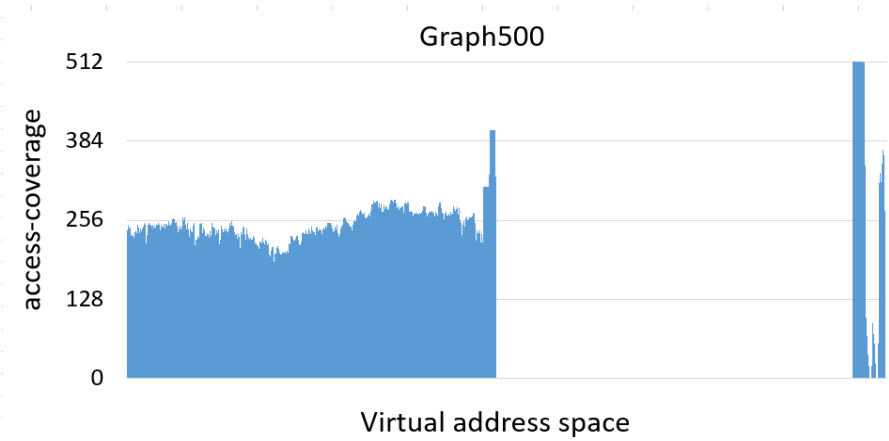
(b) Time saved per huge page promotion

Figure 3.5: Performance speedup (top sub-figure) and time saved per huge page promotion (bottom sub-figure) over baseline pages.

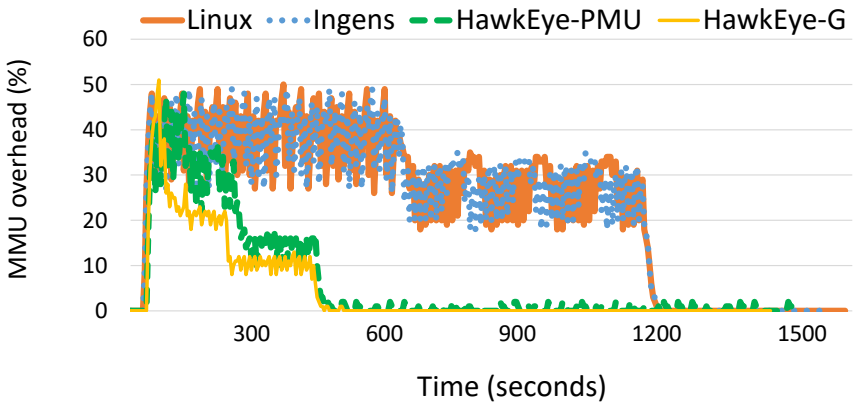
access-coverage in both performance and fairness for single, multiple homogeneous, and multiple heterogeneous workloads; (b) the bloat-vs-performance trade-off; (c) the impact of low latency page faults; (d) cache interference caused by asynchronous pre-zeroing thread; and (e) the impact of memory efficiency enabled by asynchronous page pre-zeroing.

3.4.1 Performance advantages of fine-grained huge page promotion

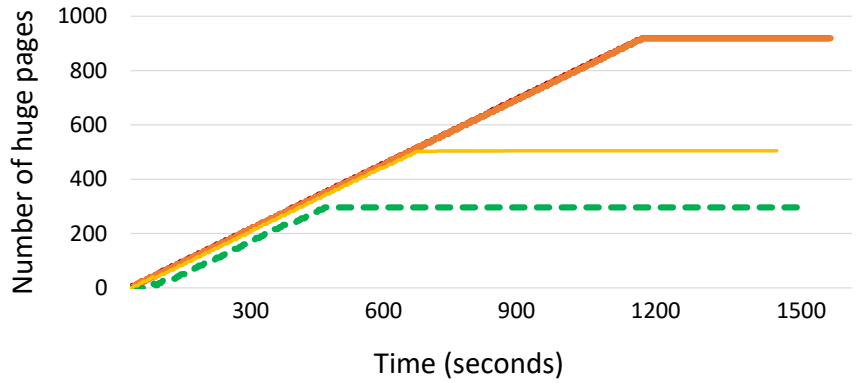
We first evaluate the effectiveness of our access-coverage based promotion strategy (see Figure 3.5. For this, we measure the time required for our algorithm to recover from a fragmented state with high address translation overheads to a state with low address translation overheads. Recall that HawkEye promotes pages based on access-coverage while previous approaches promote pages in virtual address order (from low to high). We fragment the memory initially by reading several files in memory; our test workloads are started in the fragmented state and we



(a) Access-coverage in the virtual address regions

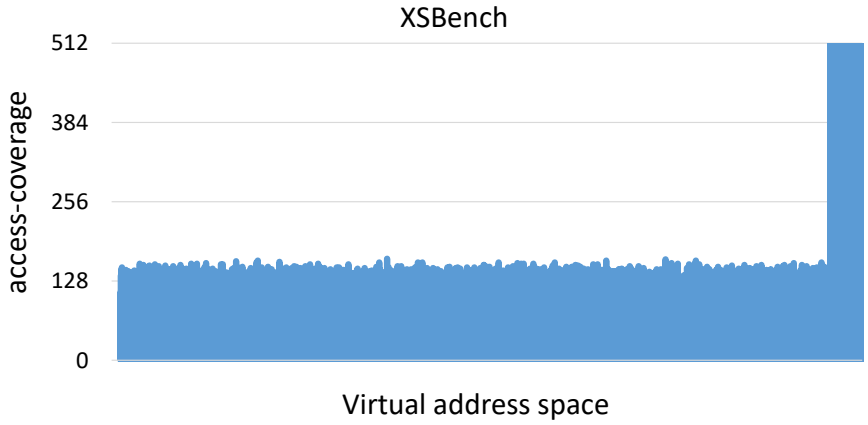


(b) MMU overhead over time

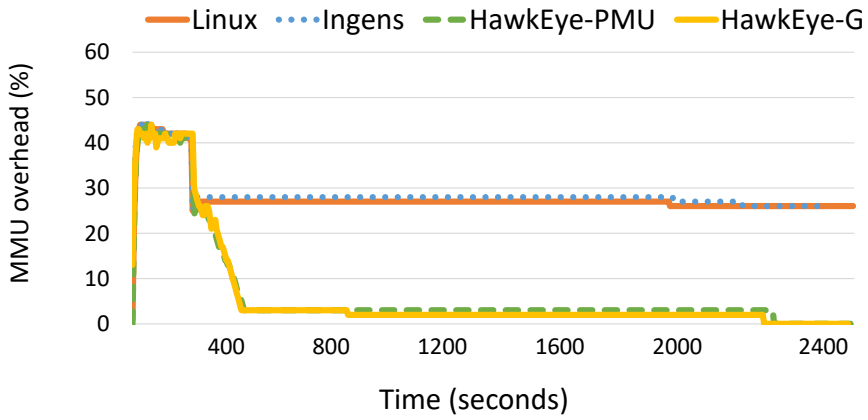


(c) Number of huge page promotions

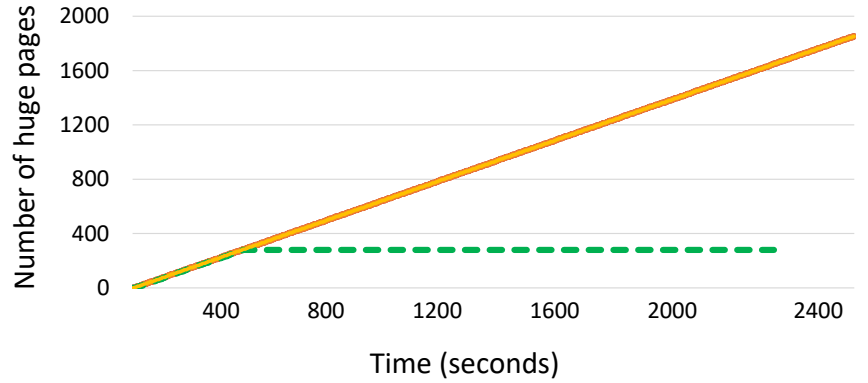
Figure 3.6: Access-coverage in application virtual address space, MMU overhead and the number of huge page promotions for `Graph500`. HawkEye reduces MMU overhead much faster and with fewer huge pages than Linux and Ingens.



(a) Access-coverage in the virtual address regions



(b) MMU overhead over time



(c) Number of huge page promotions

Figure 3.7: Access-coverage in application virtual address space, MMU overhead and the number of huge page promotions for `XSBench`. HawkEye reduces MMU overhead much faster and with fewer huge pages than Linux and Ingens.

measure the time taken for the system to recover from high MMU overheads. This experimental setup simulates expected realistic situations where the memory fragmentation in the system fluctuates over time.

Without huge page promotion, our test workloads would keep incurring high address translation overheads. We use this configuration as the baseline system. HawkEye is quickly able to recover from these overheads through appropriate huge page allocations. Figure 3.5a shows the performance improvement obtained by HawkEye (over the baseline strategy that never promotes pages) for three workloads: `Graph500`, `XSBench` and `cg.D`. These workloads allocate all their required memory in the beginning, i.e., in the fragmented state of the system. For these workloads, the speedups due to effective huge page management through HawkEye are as high as 22%. Compared to Linux and Ingens, access-coverage based huge page promotion strategy of HawkEye improves performance by 13%, 12% and 6% over both Linux and Ingens.

To understand this more clearly, Figure 3.6 shows the access pattern, MMU overheads and the number of huge pages promoted over time for `Graph500` (see Figure 3.6a, Figure 3.6b, and Figure 3.6c). Figure 3.7 shows the same for `XSBench` (see Figure 3.7a, Figure 3.7b, Figure 3.7c).

Figure 3.6a and Figure 3.7a clearly show that TLB hot-spots in these applications are concentrated in the higher end of virtual address space. Therefore, a sequential-scanning based promotion is likely to be sub-optimal because it will take a significant time to reach the most TLB sensitive regions. Figure 3.6b and Figure 3.7b corroborate this observation: for example, both HawkEye variants take ≈300 seconds to eliminate MMU overheads of `XSBench` while Linux and Ingens have high overheads even after 1000 seconds of execution. Figure 3.6c and Figure 3.7c show that HawkEye achieves these benefits with similar or fewer huge pages than Linux and Ingens.

To quantify the cost-benefit analysis of huge page promotions further, we propose a new metric: the average execution time saved (over using only baseline pages) per huge page promotion. A scheme that maximizes this metric would be most effective in reducing MMU overheads. Figure 3.5b shows that HawkEye performs significantly better than Linux and Ingens on this metric. The difference between the efficiency of HawkEye-PMU and HawkEye-G is also evident: HawkEye-PMU is more efficient as it stops promoting huge pages when MMU overheads are below a certain threshold (2% in our experiments). In summary, HawkEye-G and HawkEye-PMU are up to 6.7× and 44× more efficient (for `XSBench`) than Linux in terms of time saved per huge page promotion.

For other workloads that we evaluate in this chapter, we find that their access patterns are spread uniformly over the virtual address space. Therefore, HawkEye does not provide any

Workload	Execution Time (seconds)				
	Linux-4KB	Linux-2MB	Ingens	HawkEye-PMU	HawkEye-G
Graph500-1	2270	2145(1.06)	2243(1.01)	1987(1.14)	2007(1.13)
Graph500-2	2289	2252(1.02)	2253(1.02)	1994(1.15)	2013(1.14)
Graph500-3	2293	2293(1.0)	2299(1.00)	2012(1.14)	2018(1.14)
Average	2284	2230(1.02)	2265(1.01)	1998(1.14)	2013(1.13)
XSBench-1	2427	2415(1.0)	2392(1.01)	2108(1.15)	2098(1.15)
XSBench-2	2437	2427(1.0)	2415(1.01)	2109(1.15)	2110(1.15)
XSBench-3	2443	2455(1.0)	2456(1.00)	2133(1.15)	2143(1.14)
Average	2436	2432(1.0)	2421(1.00)	2117(1.15)	2117(1.15)

Table 3.5: Execution time of 3 instances of `Graph500` and `XSBench` when executed simultaneously. Values in parentheses represent speedup over baseline pages.

significant benefits for these applications and behaves similar to Linux and Ingens.

3.4.2 Fairness advantages of fine-grained huge page promotion

We next experiment with multiple applications to study both performance and fairness, first with identical applications, and then with heterogeneous applications, running simultaneously.

Identical workloads: To understand the effect of different huge page promotion policies on identical workloads, we perform two experiments. In the first experiment, we execute three identical instances of `Graph500`, launching them one after the other with a few seconds gap in between. Our second experiment does the same with `XSBench`. To distinguish different instances of the same workload, we label them based on the process creation order e.g., `Graph500-1` denotes the first instance of `Graph500` while `Graph500-2` and `Graph500-3` denote the second and third instances.

Figure 3.8 and Figure 3.9 show the MMU overheads and huge page allocations of all three instances of `Graph500`, respectively. Figure 3.10 and Figure 3.11 show the same for `XSBench`. Table 3.5 shows the execution time of each workload instance of `Graph500` and `XSBench` for the same experiment.

It is evident that Linux creates performance imbalance by promoting huge pages in one process at a time (based on the process arrival order). It first allocates huge pages to the first instance of the workloads, and moves to the next workload only after promoting the entire address space in the prior process. Its effect on the performance in quite visible for `Graph500` where the MMU overhead of `Graph500-1` (the first instance) drops in about 20 minutes, and that of the second instance `Graph500-2` in about 38 minutes. Unlike Linux, Ingens promotes huge pages proportionally in all three instances. However, it fails to improve the performance

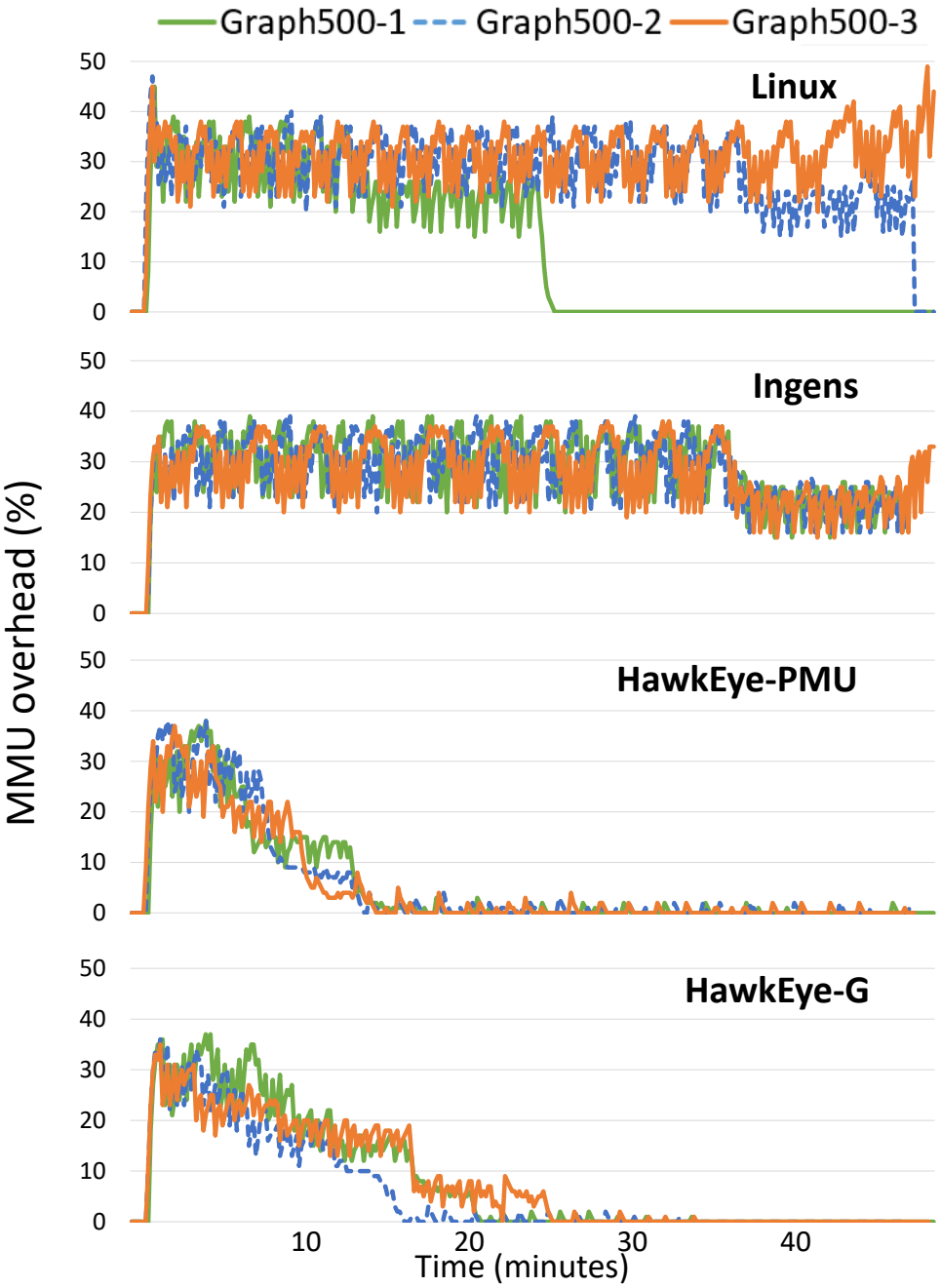


Figure 3.8: MMU overheads of three identical instances of `Graph500` while running simultaneously. Linux allocates huge pages in the order of process creation, and therefore MMU overheads reduce in the same order. Ingens allocates huge pages fairly but takes longer to reduce MMU overhead as it allocates many huge pages in TLB insensitive regions for these applications (low virtual addresses). HawkEye uses hardware performance counters and access-coverage based huge page promotion and is therefore more efficient than Linux and Ingens.

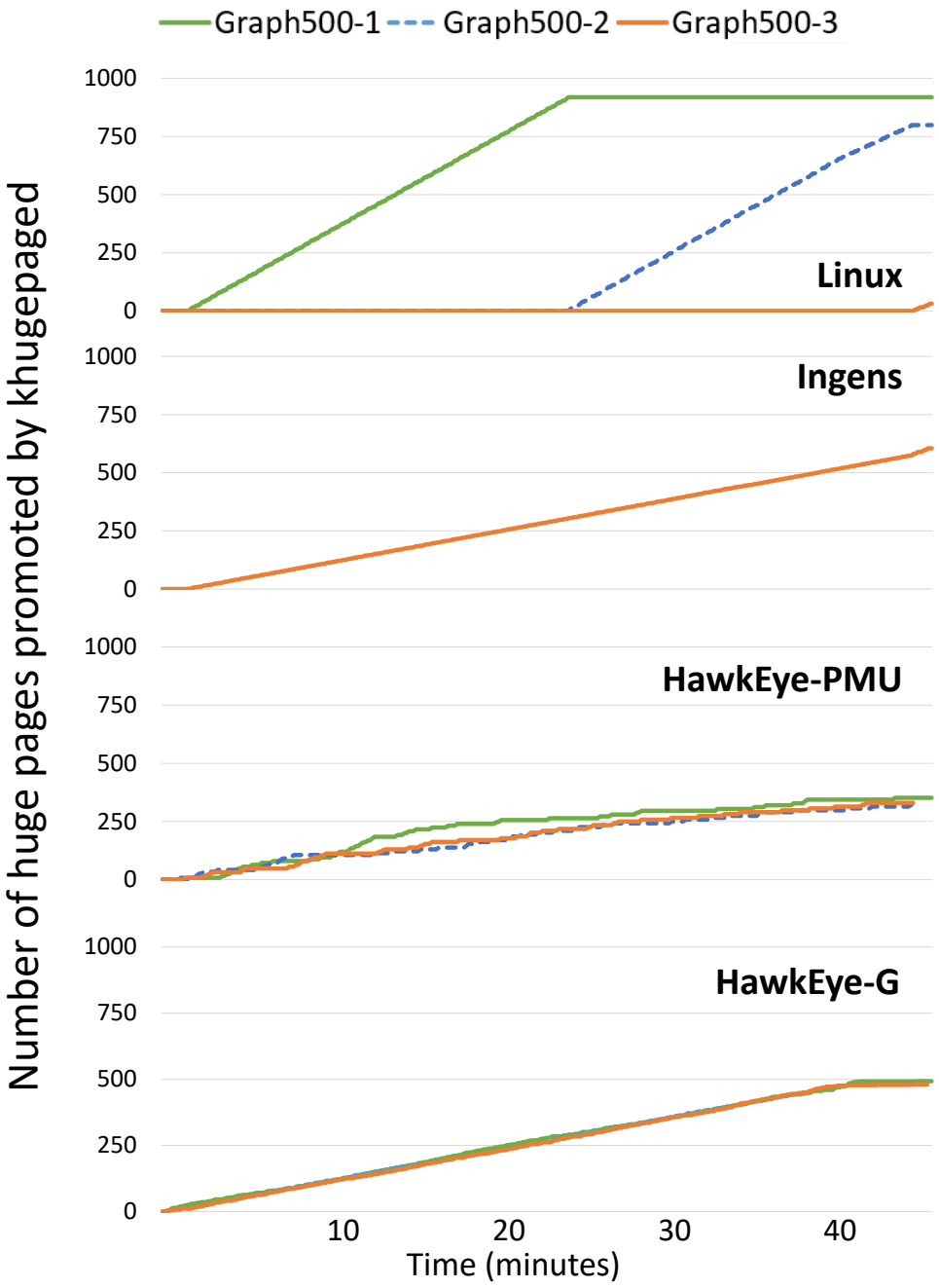


Figure 3.9: Number of huge pages promoted for all three instances of `Graph500` over time. Linux promotes huge pages in the order of process creation (e.g., `Graph500-1` followed by `Graph500-2`, and so on). Ingens promotes huge pages to all instances at the same rate (overlapping lines not visible in the graph). HawkEye also promotes huge pages at roughly similar rate to all instances, but based on the estimated benefits of allocation.

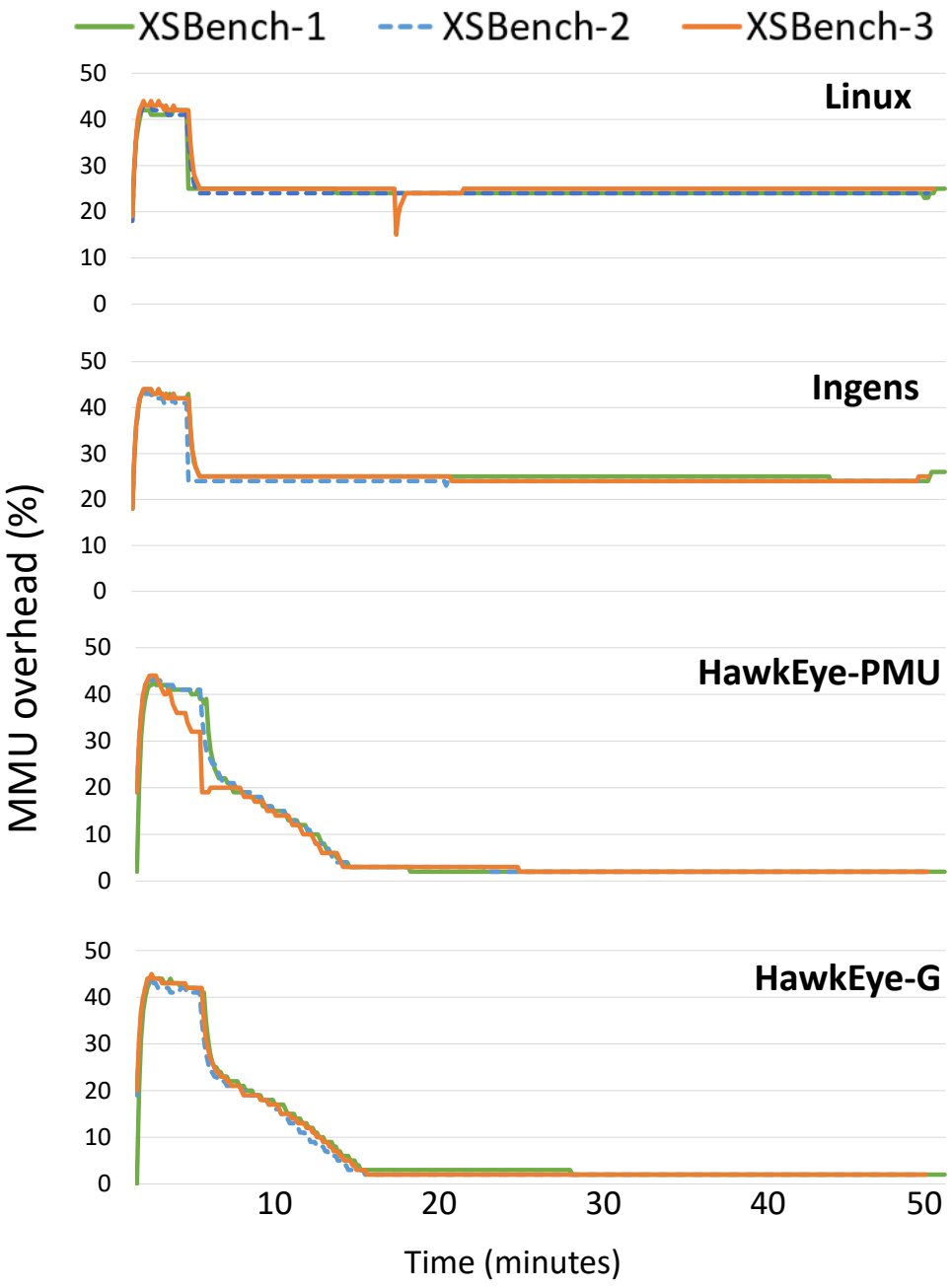


Figure 3.10: MMU overheads of three identical instances of `XSBench` while running simultaneously. Linux and Ingens takes a long time to reach the most TLB sensitive regions (high virtual addresses). Therefore, they are unable to reduce MMU overheads in this case. HawkEye uses hardware performance counters and access-coverage based huge page promotion and is therefore more efficient than Linux and Ingens.

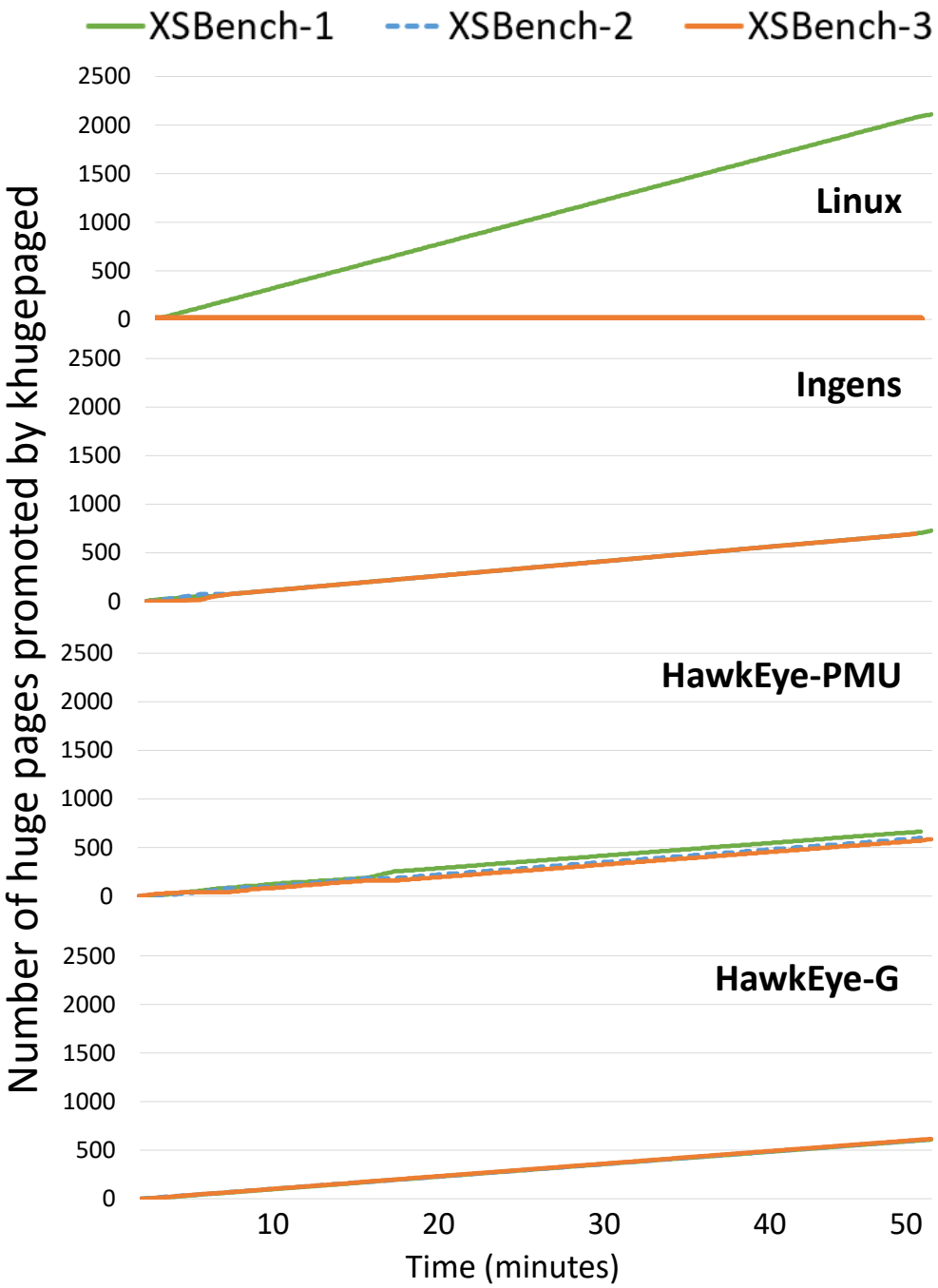


Figure 3.11: Number of huge pages promoted for all three instances of `XSBench` over time. Linux promotes huge pages in the order of process creation. Ingens promotes huge pages to all instances at the same rate. HawkEye also promotes huge pages at roughly similar rate to all instances, but based on the estimated benefits of huge pages (note that overlapping lines are not clearly visible for Ingens and HawkEye-G.

of these workloads. In fact, it may lead to poorer performance than Linux, as is visible for `Graph500-1` in Table 3.5. We explain this behaviour with an example.

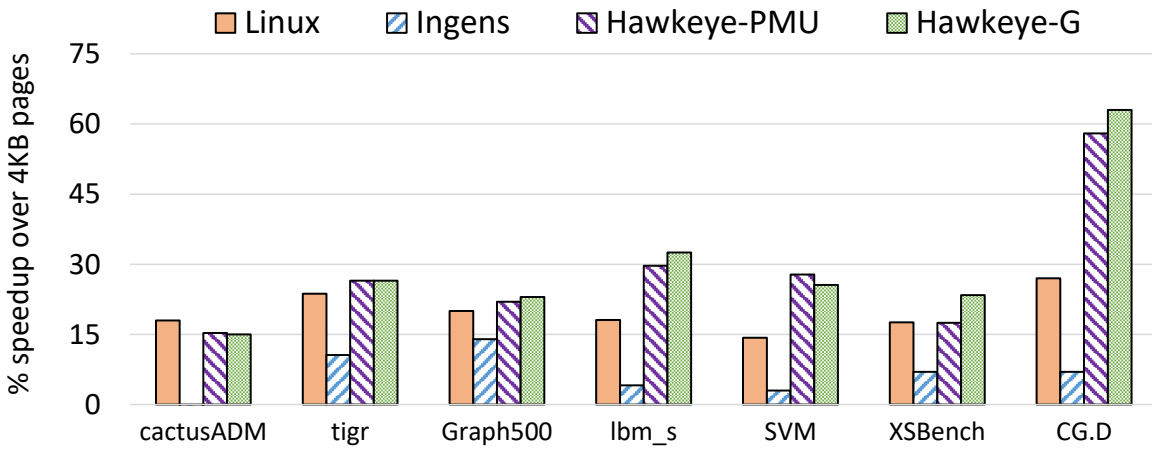
Linux selects `Graph500-1`'s address space first and promotes all pages in it in around 20 minutes. Even though this strategy is unfair, it improves the performance of `Graph500-1` because the TLB-sensitive region from the higher end of `Graph500-1`'s address space is also promoted. Linux then selects `Graph500-2` whose MMU overheads decrease after ≈38 minutes. In contrast, Ingens promotes huge pages from lower virtual address regions in each instance of `Graph500`. Thus it spends considerable time in promoting huge pages that are not really beneficial – note that the most TLB sensitive regions are towards the higher end of the virtual address space in `Graph500` and `XSBench`. This delays the promotion of huge pages in TLB sensitive regions for our workloads. Ultimately, this behavior leads to 5% performance degradation for Ingens, as compare to Linux for `Graph500-1`. For `XSBench`, the performance of both Ingens and Linux is similar (but inferior to HawkEye) because both fail to promote the application's TLB sensitive regions before the application finishes.

HawkEye ensures fairness and efficiency by judiciously distributing huge pages across all workload instances, and promoting huge pages for the most TLB sensitive regions within each address space. For example, HawkEye-PMU reduces the MMU overhead of all three instances of `Graph500` within 15 minutes whereas the same workloads continue to execute with high MMU overheads even after 30 minutes in Linux and Ingens(see Figure 3.8). Overall, HawkEye-PMU and HawkEye-G achieve 1.12× and 1.11× speedup for `Graph500`, and 1.15× speedup for `XSBench` over Linux, on average (see Table 3.5).

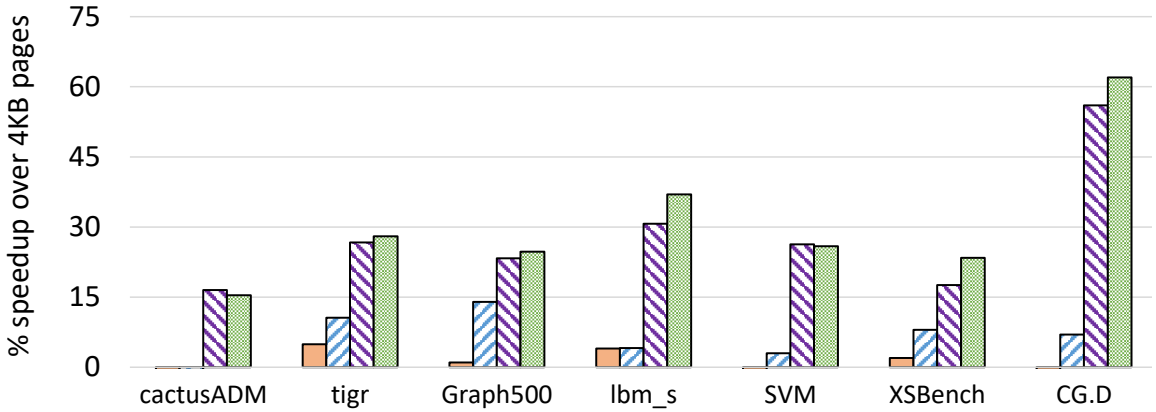
Heterogeneous workloads: To measure the efficacy of different strategies for heterogeneous workloads, we execute workloads by grouping them into sets where a set contains one TLB sensitive and one TLB insensitive application. Each set is executed twice, after fragmenting the system, to assess the impact of huge page policies on the order of execution. For a TLB insensitive workload, we execute a lightly loaded `Redis` server with 1KB-sized 40M keys servicing 10K requests per-second. Since `Redis` incurs negligible MMU overhead in this case, an efficient system is expected to allocate huge pages to co-running TLB-sensitive workloads.

The combined memory footprint of workloads in each set often exceeds 50GB, and therefore, the default promotion rate of 1.6MB per second is too low to cover the combined memory size of a set. Therefore, for this experiment, we set the huge page promotion speed to 16MB per second in all systems (i.e., Linux, Ingens and both the variants of HawkEye).

Figure 3.12 shows the performance speedup over 4KB pages. Figure 3.12a shows the configuration when TLB sensitive workloads are launched before `Redis` while Figure 3.12b shows it when TLB sensitive workloads are launched after `Redis`. These two configurations are intended



(a) when these workloads are launched **before** `Redis`



(b) when these workloads are launched **after** `Redis`

Figure 3.12: Performance speedup over baseline pages of TLB sensitive applications when they are executed alongside a lightly loaded `Redis` key-value store in different orders.

to cover two different behaviours of Linux because Linux promotes huge pages in the process launch order. Ingens and HawkEye are agnostic to process creation order.

The impact of execution order is clearly visible for Linux. In Figure 3.12a, Linux with THP support improves performance over baseline pages by promoting huge pages in TLB sensitive applications. However, Figure 3.12b, Linux promotes huge pages in `Redis` resulting in poor performance for TLB sensitive workloads. While the execution order does not matter in Ingens, its proportional huge page promotion strategy is biased to allocate huge pages in large memory workloads (`Redis` in this case). Further, our client requests randomly selected keys

Configuration	Details
HawkEye-Host	Two VMs each with 24 vCPUs and 48GB memory. VM-1 runs `Redis`. VM-2 runs TLB sensitive workloads.
HawkEye-Guest	Single VM with 48 vCPUs and 80GB memory running `Redis` and TLB sensitive workloads
HawkEye-Both	Two VMs each with 24 vCPUs. VM-1 (30GB) runs `Redis`. VM-2 (60GB) runs both `Redis` and TLB sensitive workloads

Table 3.6: Experimental setup for configurations used to evaluate a virtualized system.

which makes the `Redis` server access all its huge pages uniformly – avoiding idle huge page penalty in Ingens. Consequently, Ingens promotes most huge pages in `Redis` leading to sub-optimal performance of TLB sensitive workloads. In contrast, HawkEye promotes huge pages based on MMU overhead measurements (HawkEye-PMU) or access-coverage based estimation of MMU overheads (HawkEye-G). This leads to 15–60% performance improvement over 4KB pages irrespective of the order of execution.

3.4.3 Performance in virtualized systems

For a virtualized system, we use the KVM hypervisor running with Ubuntu16.04 and fragment the system prior to running the workloads. Evaluation is presented for three different configurations, i.e., when HawkEye is deployed at the host, guest and both the layers. Each of these configurations requires running the same set of applications differently. For example, while considering the effect of applying HawkEye in a single virtual machine, we only need a single VM running multiple workloads. However, when assessing HawkEye's benefits at the hypervisor level, we need to deploy two virtual machines on the same physical host. Similarly, when deploying HawkEye at both the layers, we run two virtual machines with multiple workloads. Table 3.6 details the configurations we used for this experiment.

Figure 3.13 shows that HawkEye provides 18–90% speedup compared to Linux in virtual environments. Notice that in some cases (e.g., `cg.D`), performance improvement is much higher in a virtualized system compared to bare-metal. This is expected since the MMU overheads are higher under virtualization – due to nested page table walks involved in servicing TLB misses of guest applications. This presents higher performance improvement opportunities that are effectively exploited by HawkEye.

3.4.4 Bloat vs. performance

We next study the relationship between memory bloat and performance; we revisit an experiment similar to the one summarized in Subsection 3.2.1. Ingens authors have also used a similar

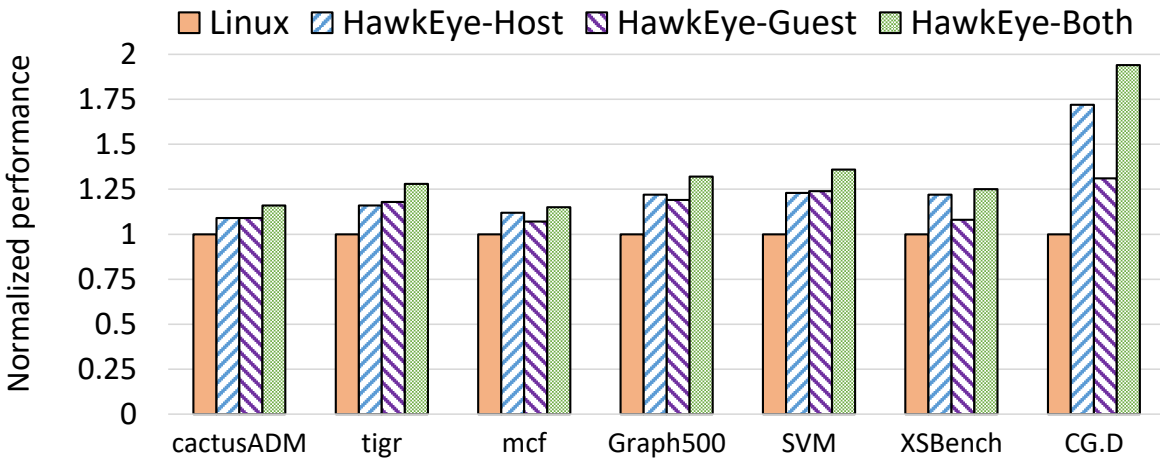


Figure 3.13: Performance compared to Linux in a virtualized system when HawkEye is applied at the host, guest and both layers.

Kernel	**Self-tuning**	**Memory Size**	**Throughput**
Linux-4KB	No	16.2GB	106.1K
Linux-2MB	No	33.2GB	113.8K
Ingens-90%	No	16.3GB	106.8K
Ingens-50%	No	33.1GB	113.4K
HawkEye (no mem. pressure)	Yes	33.2GB	113.6K
HawkEye (mem. pressure)	Yes	16.2GB	105.8K

Table 3.7: Memory consumption and throughput of `Redis` key-value store with different huge page management systems.

methodology for such an experiment [121]. We populate a `Redis` instance with 8M (10B key, 4KB value) key-value pairs and then delete 60% randomly selected keys (see Table 3.7). While Linux-4KB (no THP) is memory efficient (no bloat), its performance is low (7% lower throughput than Linux-2MB, i.e., with THP). Linux-2MB delivers high performance but consumes more memory which remains unavailable to the rest of system even when memory pressure increases. Ingens can be configured to balance the memory vs. performance trade-off. For example, Ingens-90% (Ingens with 90% utilization threshold) is memory efficient while Ingens-50% (Ingens with 50% utilization threshold) favors performance and allows more memory bloat. In either case, it is unable to recover from bloat that may have already been generated (e.g., during its aggressive phase).

By automatically switching from aggressive huge page allocations under low memory pressure to a memory conserving strategy at runtime, HawkEye is able to operate in both situations

Workload	OS Configuration				
	Linux 4KB	Linux 2MB	Ingens 90%	HawkEye 4KB	HawkEye 2MB
Redis (45GB)	233	437	192	236	551
SparseHash (36GB)	50.1	17.2	51.5	46.6	10.6
HACC-IO (6GB)	6.5	4.5	6.6	6.5	4.2
JVM Spin-up (36GB)	37.7	18.6	52.7	29.8	1.37
KVM Spin-up (36GB)	40.6	9.7	41.8	30.2	0.70

Table 3.8: Performance implications of asynchronous page zeroing. Values for `Redis` represent throughput (higher is better); all other values represent time in seconds (lower is better).

as per the expectation: (1) low MMU overheads with high memory footprint, and (2) low memory footprint at the expense of high MMU overhead, depending on the state of the system. This shows that HawkEye can navigate the trade-off space automatically at runtime.

3.4.5 Fast page faults with async page pre-zeroing

Table 3.8 shows the performance of different strategies; for the workloads chosen for these experiments, their performance depends on the efficiency of the OS page fault handler. We measure `Redis` throughput when 2MB-sized values are inserted. `SparseHash` [25] is a hash-map library in C++ while `HACC-IO` [23] is a parallel IO benchmark used with an in-memory file system. We also measure the spin-up time of two virtual machines: KVM and a Java Virtual Machine (JVM) where both are configured to allocate all memory during initialization.

Performance benefits of async page pre-zeroing with base pages (HawkEye-4KB) are modest: in this case, the other page fault overheads (apart from zeroing) dominate performance. However, page-zeroing cost becomes significant with huge pages. Consequently, HawkEye with huge pages (HawkEye-2MB) improves the performance of `Redis` and `SparseHash` by 1.26× and 1.62× over Linux. The spin-up time of virtual machines is purely dominated by page faults. Hence we observe a dramatic reduction in the spin-up time of VMs with async pre-zeroing of huge pages: 13.8× and 13.6× over Linux. Notice that without huge pages (only base pages), this reduction is only 1.34× and 1.26×. All these workloads have high spatial locality of page faults and hence the Ingens strategy of utilization-based promotion has a negative impact on performance due to a higher number of page faults.

Async pre-zeroing enables memory sharing in virtualized environments: Finally, we note that async pre-zeroing inside the physical memory of virtual machines (VMs) enables memory sharing across VMs: the free memory of a VM returns to the host through pre-zeroing in the VM and same-page merging at the host. This can have the same net effect as

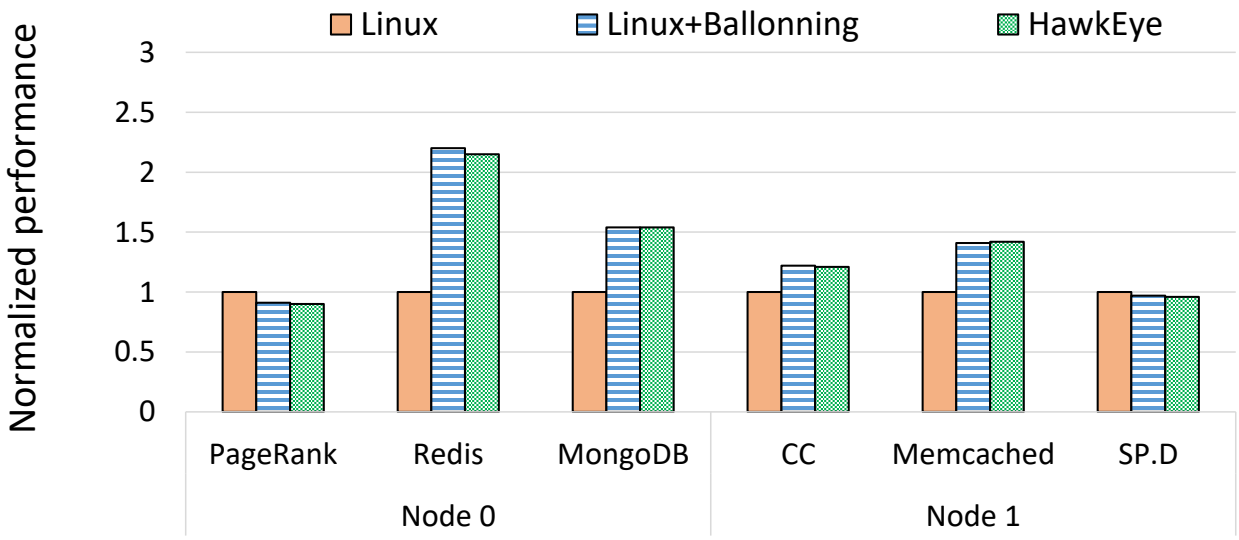


Figure 3.14: Performance normalized to the case of no ballooning in an overcommitted virtualized system.

ballooning, as the free memory in a VM gets shared with other VMs. In our experiments with memory over-committed systems, we have confirmed this behaviour: the overall performance of an overcommitted system, where the VMs are running HawkEye, matches the performance achieved through para-virtual interfaces like memory ballooning.

For example, in an experiment involving a mix of latency-sensitive key-value stores and HPC workloads (see Figure 3.14) with a total peak memory consumption of around 150GB (1.5× of total memory), HawkEye-G provides 2.3× and 1.42× higher throughput for `Redis` and `MongoDB` along with significant tail latency reduction. For `PageRank`, performance degrades slightly due to a higher number of COW page faults due to unnecessary same-page merging.

These results are very close to the results obtained when memory ballooning is enabled on *all* VMs. We believe that this is a potential positive of our design and may offer an alternative method to efficiently manage memory in over-committed systems. It is well-known that balloon-drivers and other para-virtual interfaces are riddled with compatibility problems [45], and a fully-virtual solution to this problem would be highly desirable. While our experiments show early promise in this direction, we leave an extensive evaluation for future work.

3.4.6 Performance overheads of HawkEye

We extensively evaluated HawkEye in non-fragmented systems, and with several workloads that don't benefit from huge pages. Our observations suggest that the performance of HawkEye is

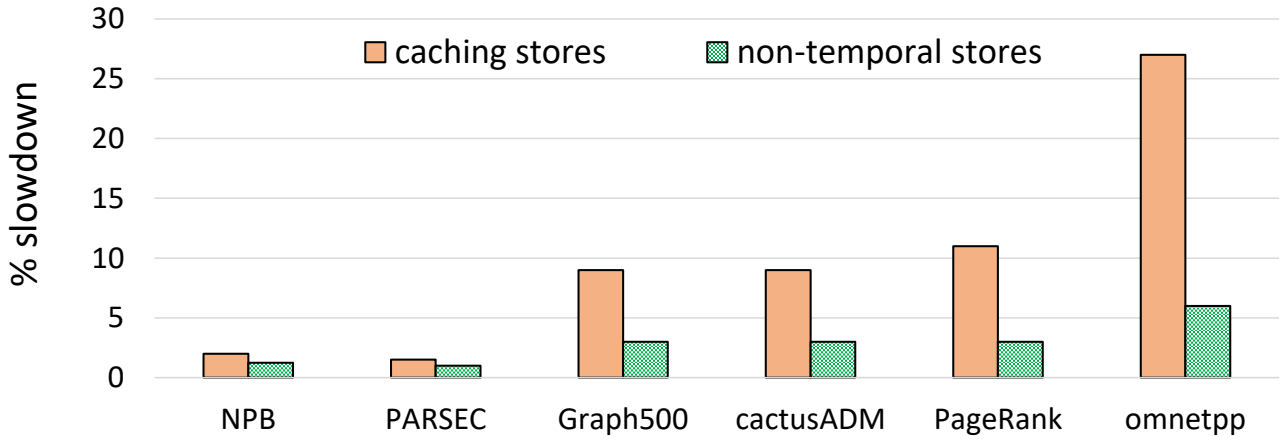


Figure 3.15: Performance overhead of async pre-zeroing with and without caching instructions. The first two bars (NPB and Parsec) represent the average of all workloads in the respective benchmark suite.

very similar to that of Linux or Ingens in these scenarios, confirming that HawkEye's algorithms for access-bit tracking, asynchronous huge page promotion and memory de-duplication add negligible overheads over existing mechanisms in Linux. However, the async pre-zeroing thread requires special attention as it may become a source of noticeable performance overhead for certain types of workloads, as discussed next.

Overheads of async pre-zeroing: A primary concern with async pre-zeroing is its potential detrimental effect due to memory and cache interference (see Subsection 3.3.1). Recall that we employ memory stores with non-temporal hints to avoid these effects. To measure the worst-case cache effects of async pre-zeroing, we run our workloads while simultaneously zero-filling pages on a separate core sharing the same L3 cache (i.e., on the same socket) at a high rate of 0.25 million 4KB pages per second (1GB per second) with and without non-temporal memory stores. For this experiment, we evaluated more than 60 workloads, including all application from the SPEC CPU 2006, GAPBS, PARSEC and NAS Parallel Benchmark suites [62, 56, 49, 105]. We report numbers for workloads that experiences the highest interference due to async page pre-zeroing thread.

Figure 3.15 shows that using non-temporal hints significantly brings down the overhead of async pre-zeroing (e.g., from 27% to 6% for `omnetpp`): the remaining overhead is due to additional memory traffic generated by the pre-zeroing thread. Further we note that this experiment presents a highly-exaggerated behavior of async pre-zeroing on worst-case workloads. In practice, the zeroing thread is rate-limited (e.g., at most 10k pages per second), and the

Workload	MMU Overhead	Time (seconds)		
		4KB	HawkEye-PMU	HawkEye-G
random(4GB)	60%	582	328(1.77×)	413(1.41×)
sequential(4GB)	< 1%	517	535	532
Total		1099	863(1.27×)	945(1.16×)
cg.D(16GB)	39%	1952	1202(1.62×)	1450(1.35×)
mg.D(24GB)	< 1%	1363	1364	1377
Total		3315	2566(1.29×)	2827(1.17×)

Table 3.9: Comparison between HawkEye-PMU and HawkEye-G for two sets of workloads. Values in parentheses represent speedup over baseline pages (whereever significant).

expected cache-interference overheads would be proportionally smaller.

3.4.7 Comparison between Hawk-PMU and HawkEye-G

Even though HawkEye-G is based on simple and approximated estimations of TLB contention, it is reasonably accurate in identifying TLB sensitive processes and memory regions in most cases. However, HawkEye-PMU performs better than HawkEye-G in some cases. Table 3.9 demonstrates two such examples where four workloads, all with high access-coverage but different MMU overheads, are executed in two sets: each set contains one TLB sensitive and one TLB insensitive workload.

The 4KB column represents the case where no huge pages are promoted. As discussed earlier in Subsection 3.2.4, MMU overheads depend on the complex interaction between the hardware and access pattern. In this case, despite having high access-coverage in their virtual address regions, `sequential` and `mg.D` have negligible MMU overheads (i.e., they are TLB insensitive). While HawkEye-PMU correctly identifies the process with higher MMU overheads for huge page allocation, HawkEye-G treats them similarly (due to imprecise estimations) and may allocate huge pages to the TLB insensitive process. Consequently, HawkEye-PMU performs up to 36% better than HawkEye-G. Bridging this gap between the two approaches in a hardware independent manner is an interesting future work.

3.5 Summary

To summarize, transparent huge page management is essential but complex. Effective and efficient algorithms are required in the OS to automatically balance the trade-offs and provide performant and stable system behavior. We expose some important subtleties related to huge page management in existing proposals, and propose a new set of algorithms and policies to address them in HawkEye.

Chapter 4

Leveraging Architectural Support for All Page Sizes

In Chapter 2 and Chapter 3, we address various memory management challenges involved in transparent huge page management. Unfortunately, transparent huge pages support only 2MB pages in current system software wherein high-end servers support 100s of gigabytes to a few terabytes of physical memory today. We find that for many applications, 2MB huge pages fail to satisfactorily mitigate the overhead of virtual-to-physical address translation. In this chapter, we propose multi-level transparent huge pages to benefit from the hardware support for all page sizes. We show that very large pages (e.g., 1GB pages available on x86 architecture) are essential for efficiently supporting virtual memory on modern systems.

4.1 Introduction

It is not uncommon for hardware features to require software enablement. Unfortunately, it is also common to find the micro-architectural resources to remain underutilized due to the lack of adequate software support. One pays for the underutilized hardware through both – the runtime cost (e.g., power dissipation), and the design and verification cost. Further, architects are left in the dark about the extent to which those features are beneficial to applications in practice, and whether they should continue enhancing them or drop them in future products.

In this chapter, we shed light on one such hardware feature – 1GB pages, that has been mostly languishing for a decade due to inadequate system software (here, Linux and KVM) enablement. The x86-64 processors have long supported two large page sizes – 2MB and 1GB, for over a decade. Intel's Sandybridge launched in 2010 first supported 1GB pages and had a four-entry L1 TLB dedicated for 1GB pages in each core [38]. Since then, both large page

sizes have found patronage of the processor vendors. Recent Cascade Lake processors support 32-entries in L1 TLB and up to 1536 entries in L2 TLB for 2MB pages. It also has 4 and 16 entries for 1GB pages, in L1 and L2 TLB respectively [37]. The lastest Ice Lake Xeon processors can hold up to 1024 entries each for both 2MB and 1GB pages, per core, in its L2 TLBs [46].

While hardware vendors continue to enhance TLB capacities for both large page sizes; more so for 1GB pages in recent times, the system software – operating systems and hypervisors – continues to focus only on 2MB large pages. For example, Linux's Transparent Huge Pages (THP) for application-transparent dynamic allocation of large pages limits itself to only 2MB pages.

It is thus pertinent to wonder if micro-architects are justified in continuing to invest hardware resources for both 1GB and 2MB large pages. Our first contribution is an empirical analysis to answer the above question. We quantify the usefulness of 1GB pages, *over and above* 2MB pages, to various applications, with and without virtualization. We find that while most memory-intensive applications benefit from 2MB pages over 4KB, a subset of them speeds up further with 1GB pages. Besides our own analysis, Google recently reported that nearly 20% of CPU cycles across its data centers are attributed to serving TLB misses, even after deploying 2MB large pages [106]. Further, the advent of denser non-volatile memory (NVM) technologies promises to significantly increase the physical memory size [130, 77]. The ability to efficiently address a large amount of memory is essential to harness its full benefit.

From a theoretical perspective, using the largest page size seems to be an ideal choice. However, practical constraints often prohibit the use of largest page size. For example, mapping a virtual address range with a large page requires the virtual address range to be at least as long as that page size and to be aligned at that page size boundary. Consequently, larger the page size, lesser is the number of virtual address ranges that are mappable by that page size. Further, external physical memory fragmentation also makes it difficult to allocate large physical memory chunks. Therefore, it is desirable to fallback to smaller large pages, if allocating the largest page size is not possible. Due to these constraints, we find that using multiple large page sizes in tandem is important.

This chapter details the design and implementation of Trident [*] – a system that dynamically allocates all available page sizes in x86-64 processors to fully harness the processor's TLB resources. Trident does not require prior reservation of physical memory or application modifications. A key challenge in dynamically allocating 1GB pages is the hardship in ensuring availability of 1GB contiguous physical memory chunks when needed. As the free physical memory gets naturally fragmented over time, finding 1GB chunks becomes more difficult than 2MB

[*]The name draws from the fact that Trident uses three page sizes (i.e., 1GB, 2MB and 4KB).

chunks. Thus, the dynamic allocation of large pages needs to periodically compact physical memory for making free memory contiguous. However, compaction for 1GB memory requires significantly more work than 2MB. Moreover, a compaction attempt fails if it encounters even a single page frame with unmovable contents, e.g., kernel objects like inodes, in a 1GB region (as discussed in Chapter 2). In short, compaction at the granularity of 1GB pages needs a new approach.

Trident introduces a smart-compaction technique. We observe that the current approach of *sequential scanning* and moving contents of occupied page frames is not scalable to 1GB. This approach incurs a large amount of avoidable data movement. With smart-compaction, Trident tracks the number of occupied bytes (i.e., the number of mapped page frames) within each 1GB physical memory region and uses this information to minimize data movement during compaction. It also tracks unmovable pages, e.g., Linux's own data structures, within a 1GB region to avoid unnecessary data movement. If 1GB pages are still not available, Trident allocates the next best alternative i.e., 2MB pages. These 2MB page mappings are later promoted to 1GB pages, when suitable.

We also propose an extension of Trident called Trident^{pv} for virtualized systems. Trident^{pv} virtualizes data movement by replacing page-level memory copy operations with page table pointer manipulations. This copy-less technique makes the promotion of 2MB pages to a 1GB page significantly faster than the traditional copy-based approach. In this approach, the guest OS and the hypervisor coordinate to alter the desired guest physical address (gPA) to host physical address (hPA) mappings.

Overall, Trident speeds up eight memory-intensive applications by 18%, over Linux's THP, on average. Trident^{pv} further improves performance under virtualization, by up to 10%.

In summary, this chapter makes the following contributions:

- We evaluate the usefulness of 1GB pages across various applications, both without and with virtualization.
- We empirically demonstrate why it is important to deploy all large page sizes, not only the largest one.
- We created Trident in Linux to dynamically allocate all page sizes available on x86-64 processors to speed up applications with large memory footprint.
- We then propose an optional extension to Trident called Trident^{pv} that employs paravirtualization to enable copy-less 1GB page promotion and compaction in the guest OS.

Processor	Intel Xeon Gold 6140 @2.3GHz Skylake Family with 2 Sockets
Number of cores	18 cores (36 threads) per socket
L1-iTLB	4KB pages, 8-way, 128 entries 2MB pages, fully associative, 8 entries
L1-dTLB	4KB pages, 4-way, 64 entries 2MB pages, 4-way, 32 entries 1GB pages, fully associative, 4 entries
L2 TLB	4KB/2MB pages, 12-way, 1536 entries 1GB pages, 4-way, 16 entries
Cache	32K L1-d, 32K L1-i, 1MB L2, 24MB L3
Main Memory	384GB (192GB per socket)
OS / Hypervisor	Ubuntu with Linux kernel version 4.17.3 / KVM

Table 4.1: Specification of the experimental system

Name	Threads	Memory	Description
XSBench	36	117GB	Monte Carlo particle transport algorithm for nuclear reactors [176]
SVM	36	67.9GB	Support Vector Machine, kdd2012 dataset [36]
Graph500	36	63.5GB	Breadth-first-search and single-source-shortest-path over undirected graphs [177]
CC/BC/PR	36	72GB	Graph algorithms from GAPBS [56]
CG.D	36	50GB	Congruent Gradient algorithm from NAS Parallel Benchmarks [49]
Btree	1	10.5GB	Random lookups in a B+tree
GUPS	1	32GB	Irregular, memory-intensive microbenchmark [20]
Redis	1	43.6GB	An in-memory key-value store [67]
Memcached	36	79GB	An in-memory key-value caching store [89]
Canneal	1	32GB	Simulated cache-aware annealing from PARSEC [62]

Table 4.2: Specifications of the benchmarks.

4.2 Methodology

Table 4.1 details the configuration of our experimental platform. We evaluate 12 multi-GB workloads, as detailed in Table 4.2, across various domains such as machine-learning, graph algorithms, key-value stores, HPC, and micro-benchmarks (GUPS and Btree). We use Linux's perf tool [10] to collect relevant micro-architectural events. Our methodology to measure address translation overhead is similar to the one discussed in Section 3.3, except for the name of micro-architectural events that changed between Intel Haswell and Intel Skylake processors. Specifically, on Skylake CPUs, we measure the number of cycles spent on page walks via hardware performance counters DTLB_LOAD_MISSES.WALK_ACTIVE and DTLB_STORE_MISSES.WALK_ACTIVE [35].

To study performance under different states of the system, we used our open source tool to fragment the physical memory [181]. Fragmentation is measured using Free Memory Fragmentation Index (FMFI [97]) that lies between 0 (no fragmentation) and 1 (full fragmentation). Our tool fragments physical memory by first caching a large file in operating system page-cache and then reading it at random offsets for 10 minutes via 24 user threads. Caching file increases FMFI to 0.95 and random accesses ensure that page reclamation frees memory pages in non-contiguous chunks.

4.3 How useful are 1GB large pages?

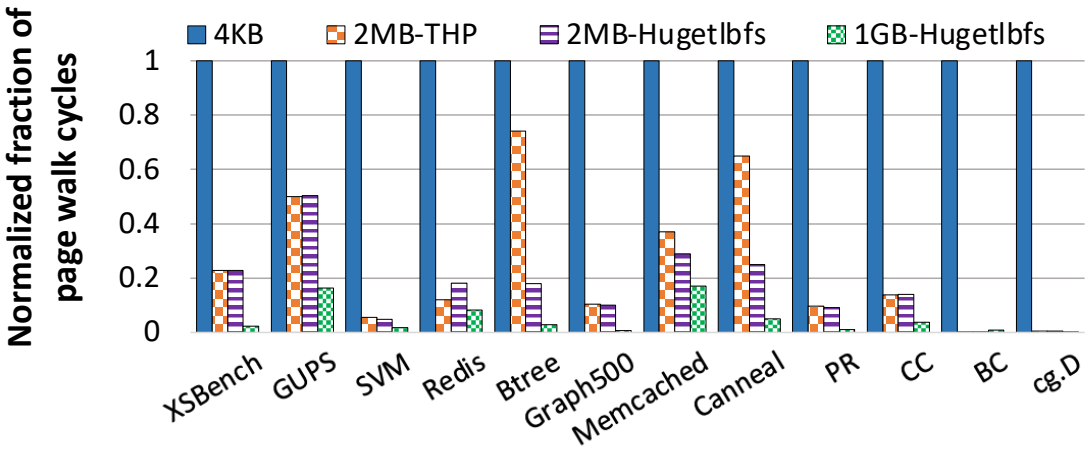
Hardware support for 1GB pages is not free, and the software running on x86 processors pays the price, irrespective of its use of 1GB pages. For example, modern Intel processors have 4-entry L1 TLB and 16-entry L2 TLB dedicated to 1GB pages. Those four L1 entries for 1GB pages are accessed on every load and store since the page size is not known in advance. Due to frequent accesses, L1 TLBs can contribute to a thermal hotspot in processors [155] and can account for 6% of a processor's total power [166]. The presence of dedicated TLBs for 1GB pages adds to the cost. The continued increase in the number of TLB entries for 1GB pages would worsen it. It is, thus, natural to wonder if applications can benefit from 1GB pages. We analyze various applications under different execution scenarios to understand the usefulness of 1GB pages.

4.3.1 1GB pages in native execution

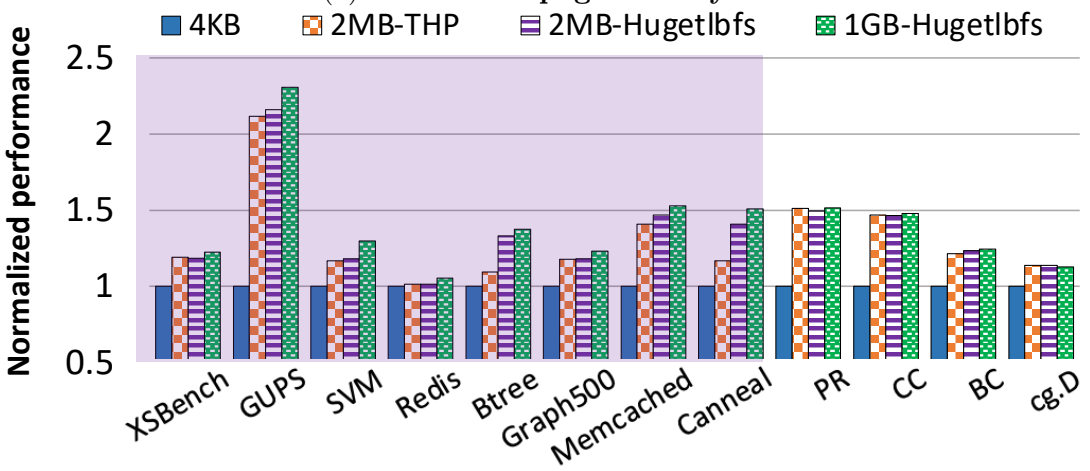
Figure 4.1a shows the normalized fraction of execution cycles spent on page walks for each application while using different page sizes. The four bars for each application represent walk cycles with ① 4KB pages, ② dynamically allocated 2MB pages via THP, ③ statically pre-allocated 2MB pages via libHugetlbfs, and, ④ statically pre-allocated 1GB pages via libHugetlbfs. The fourth bar approximates the performance achievable if the 1GB pages are deployed but not 2MB. Application-transparent dynamic allocation of 1GB pages (i.e., THP like) is not supported in Linux today.

Note that THP often performs as good as 2MB-libHugetlbfs. For `Redis`, THP reduces slightly more walk cycles than libHugetlbfs by also mapping the stack with huge pages (libHugetlbfs does not map a process's stack with huge pages). Importantly, THP does not require pre-allocation of physical memory, nor does it need users to statically decide which program segment(s) to be mapped with large pages.

Reduction in page walk cycles does not always lead to proportional performance gain on out-of-order cores. Rather, the speed up depends upon what portions of walk cycles are on the



(a) **Fraction of page-walk cycles**

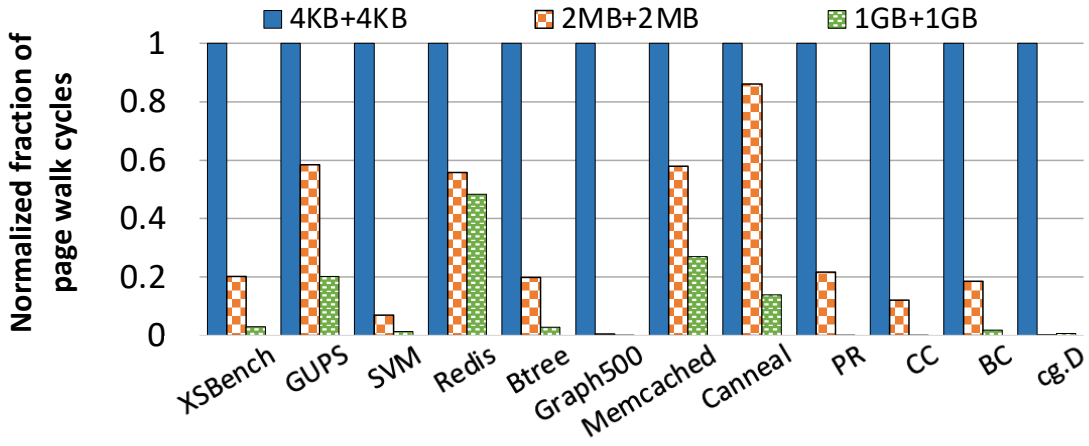


(b) **Normalized performance.**

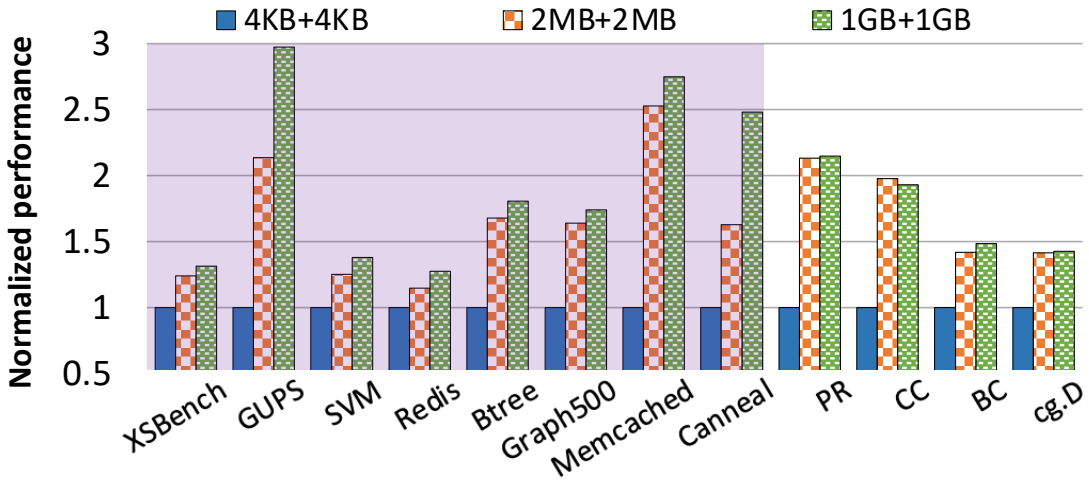
Figure 4.1: Performance impact of different page sizes under native execution. Applications in shade benefit from 1GB pages.

critical path of execution. Figure 4.1b shows the normalized performance. For all workloads, except `Redis` and `Memcached`, we measure performance in terms of their execution time. For `Redis` and `Memcached`, we report performance in terms of throughput.

We observe non-negligible performance improvement (at least 3%) for eight applications (shaded left part of the figure) with use of 1GB pages over 2MB pages. For example, `Canneal` speeds up by 30% over THP. These eight applications' performance improves by 12.5%, on average, when 1GB pages are used via libHugetlbfs, relative to THP using 2MB pages. Rest of the applications witness benefits of using 2MB pages over 4KB, but barely gain any further with 1GB pages. This is not surprising; the walk cycles were already low with 2MB pages, and an out-of-order CPU could hide the rest. Henceforth, we thus focus on the first eight (shaded)



(a) **Fraction of page-walk cycles**



(b) **Normalized performance**

Figure 4.2: Performance impact of different page sizes under virtualization. Applications in shade benefit from 1GB pages.

applications.

We also observe that with THP, applications perform within 0.5% of that with libHugetlbfs using 2MB pages, even though it does not requires memory pre-allocation or user hints. This emphasizes the importance of THP for the wide deployment of 2MB pages; something that is lacking for 1GB pages.

4.3.2 1GB pages under virtualized execution

Two levels of translations under virtualization increase overheads. Each level may use a different page size. Thus, nine combinations of page sizes are possible. While we explored all, we discuss only 4KB-4KB, 2MB-2MB, and 1GB-1GB combinations where the first and second terms denote the page size used in the guest and in the host, respectively. We chose these configurations as

they demonstrate the best performance achievable with a given page size.

Figure 4.2a shows the normalized fraction of page walk cycles under three different page size combinations. We notice significant reductions in walk cycles with 2MB and 1GB pages. For example, the fraction of walk cycles reduced by 80% for `XSBench`. Even a couple of 1GB page agnostic applications (e.g., `PR`, `CC`) experience a large reduction in walk cycles. Figure 4.2b shows the performance under virtualization. We observe that 1GB pages provide a bit more benefit here. The eight 1GB page sensitive applications speed up by 17.6% over 2MB pages, on average. The application `BC`, which did not benefit from 1GB pages under native execution, becomes slightly sensitive to 1GB pages under virtualization.

4.3.3 Importance of using all large page sizes

In the analysis so far, only one of the large page sizes was deployed as is the norm in today's software. However, we find that using all large page sizes together, can bring benefits that are not achievable using any one of them.

A virtual address range is mappable by a large page only if: ① it is at least as long as that large page, and ② the starting address is aligned at the boundary of that page size. All 1GB-mappable address ranges are, thus, mappable by 2MB pages but not vice-versa. When an application allocates, de-allocates, and re-allocates memory (e.g., `Graph500`), the virtual address space gets fragmented. Consequently, an application's entire address space may not be mappable by the largest page size.

We empirically find that often GBs of an application's virtual memory is 2MB-mappable but not 1GB-mappable. This depends on the application's memory allocation strategy – whether the application pre-allocates memory in large chunks (low virtual memory fragmentation) or incrementally allocate/de-allocate memory over time (high fragmentation in virtual memory).

We measure the size of 1GB-mappable and 2MB-mappable address regions with a kernel module that periodically scans the virtual address space of an application. Figure 4.3 shows the size of allocated virtual memory that is mappable with 2MB and 1GB over time for two representative applications – `Graph500` and `SVM`. The x-axis represents the post-initialization execution timeline, and the y-axis is the size of allocated virtual memory in GB.

The two lines in each graph show the amount of 1GB and 2MB-mappable memory. We observe that several GBs of memory is mappable by 2MB pages but not by 1GB (the gap between the two lines). If only 1GB pages are used, these memory regions have to be mapped with 4KB pages while wasting 2MB TLB resources.

We then analyzed the importance of mapping the regions that are un-mappable by 1GB pages, with 2MB. We wrote another module to measure the relative (sampled) TLB miss

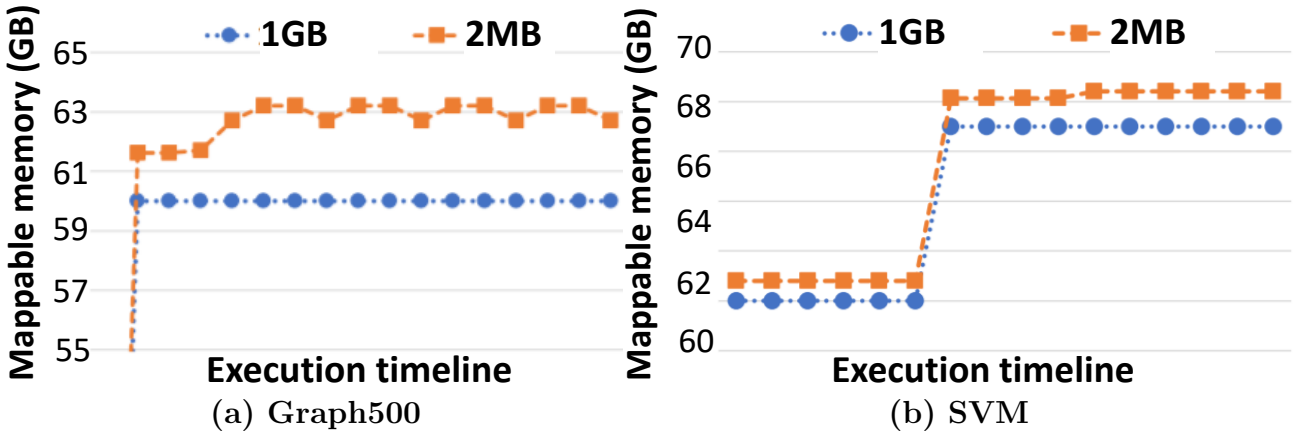


Figure 4.3: Total memory mappable with different page sizes.

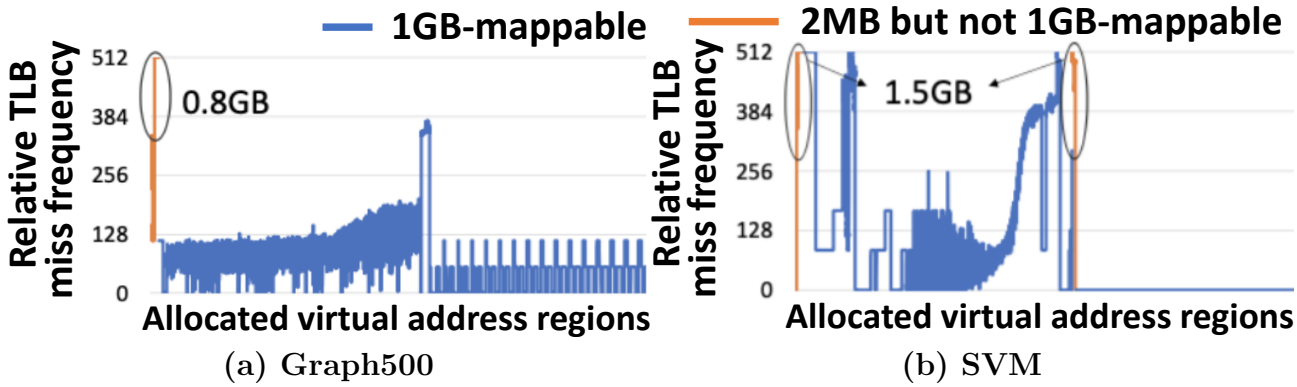


Figure 4.4: Relative TLB-miss frequency.

frequencies to the addresses that are mappable by 2MB but not by 1GB and those by both. We periodically un-set the access bits in PTEs (4KB) and then track which ones get set again by the hardware, signifying a TLB miss. Figure 4.4 presents the measurement. The x-axis shows the allocated virtual address regions, and the y-axis shows the relative TLB miss frequencies to pages in those regions. We use different colors for addresses that are 2MB-mappable but 1GB-unmappable, and those that are 1GB-mappable. We observe that the 1GB-unmappable regions witness frequent TLB misses. Particularly for `Graph500`, the spike in miss frequency on a relatively small 1GB-unmappable region (about 800MB) stands out (circled). Therefore, it is important to map these 1GB un-mappable address ranges with 2MB pages to reduce the number of TLB misses.

Furthermore, it may not always be possible to map a 1GB-mappable address range with a 1GB page due to unavailability of 1GB contiguous physical memory. However, 2MB contiguous physical memory regions are more easily available. In short, it is important to utilize all available

page sizes.

We also studied the usage of 1GB pages to Linux kernel itself. The kernel direct maps entire physical memory with the largest page size (here, 1GB). Using OS intensive workloads (e.g., apache web server and filebench [34, 174]), we found that 1GB pages improve kernel's performance by 2-3% over 2MB pages.

Summary of observations: (1) A set of niche but important big-memory applications speeds up with 1GB pages over 2MB pages. In contrast, 2MB pages universally benefit memory-intensive applications. (2) Application-transparent allocation of 2MB pages brings benefits of 2MB pages without user intervention – a capability that is lacking for 1GB pages. (3) It is important to utilize all large page sizes not only the largest.

Why the lack of 1GB software enablement? It is natural to wonder why there has not been an effort for wider enablement of 1GB pages. We hypothesize at least three reasons behind it: (1) there was no significant quantification of the usefulness of 1GB pages, over and above 2MB pages. The above analysis tries to address that gap, (2) early processor designs had a limited number of TLB entries for 1GB pages (e.g., four in Sandy Bridge). Therefore, there was apprehension about the potential thrashing of TLB if 1GB pages were used by applications with poor locality [147, 54]. However, with newer processors accommodating 1GB pages in L2 TLB, we find no evidence of such thrashing in real-world applications, and (3) finally, with the advent of denser NVM technologies and five-level page tables, the need for low-overhead address translation has never been greater. Nevertheless, as the rest of the chapter will demonstrate, managing 1GB pages in software needs special care, and thus, the bar for software enablement is non-negligible. However, given the necessity, it is imperative to eschew the software complexity for wider adoption of 1GB pages.

4.4 Trident: Dynamic allocation of all page sizes

We design and implement Trident in Linux for application-transparent dynamic allocation of all large page sizes on x86 processors without needing apriori reservation of physical memory. Trident minimizes TLB misses by mapping most of an application's address space with 1GB pages, failing which 2MB, and finally, 4KB pages are used.

Challenges: While the dynamic allocation of 2MB pages is not new, that for 1GB pages gives rise to new challenges. First, Trident needs to ensure a steady supply of free contiguous 1GB physical memory chunks even in the presence of fragmentation. We found that Linux's sequential scanning based memory compaction for creating 2MB chunks is not scalable to 1GB as it incurs excessive data copying.

Linux tracks only up to 4MB free physical memory chunks. However, the dynamic allocation

of 1GB pages would require maintaining free memory up to 1GB granularity. Allocating 1GB pages during page faults are much slower than that for 2MB or 4KB pages due to the latency of zeroing entire 1GB memory. Low-latency 1GB page fault is necessary for an aggressive deployment of 1GB pages. Finally, Trident should map a virtual address range with the largest large page size deployable at any given time. It should then periodically look for opportunities to promote address ranges mapped with a smaller large page to a larger one wherever possible.

4.4.1 Design and implementation

At a high-level, Trident modifies four major parts of Linux: (1) it enhances Linux to track up to 1GB free physical memory chunks, (2) it updates the page fault handler to allocate a 1GB page on a fault when possible and fall back to smaller pages, if needed, (3) Trident extends THP's khugepaged daemon thread to promote virtual address ranges to 1GB pages when possible, and (4) Trident employs smart-compaction technique for a steady supply of 1GB physical memory chunks at low overhead.

4.4.1.1 Managing 1GB physical memory chunks

Linux's buddy allocator keeps an array of free lists of physical memory chunks of sizes 4KB up to 4MB in the power of 2 [84]. When free memory is needed, the buddy allocator provides a memory chunk from one of its lists based on the request size. Freed physical memory is returned to the buddy, and coalesced with neighboring free memory chunks to create larger ones. Unfortunately, the buddy only keeps track of regions up to 4MB. We thus extended it to include separate lists for tracking up to 1GB memory chunks.

4.4.1.2 Allocating large pages during page fault

Like THP, Trident allocates large pages either (1) during a page fault (e.g., when a process accesses a virtual address for the first time) or (2) later during attempts to promote an address range to a large page. We here detail the former. If the faulting virtual address falls in a 1GB-mappable address range, then Trident attempts to map it with a 1GB page. If it fails, Trident attempts to map the address with a 2MB page, and on failure, with 4KB. If the faulting address falls in a region that is 2MB-mappable but not 1GB-mappable Trident tries to map it with 2MB.

Asynchronous zero-fill: A 1GB page fault takes around 400 milli-seconds compared to 450 micro-seconds for a 2MB page. The additional latency is due to zero-filling of 1GB memory instead of 2MB*. Similar to HawkEye, we employ asynchronous zero-fill to speed up 1GB faults.

*Zero-fill ensures application's leftover data does not leak out.

A kernel thread periodically zero-fills free 1GB regions and Trident allocates a zero-filled region, if available. This makes the average 1GB fault latency comparable to that of a 4KB page.

Table 4.3 and Table 4.4 show the portion of applications' memory footprints mapped by 1GB and 2MB pages under Trident's various allocation mechanisms that we will discuss in this section. The first two data columns in each table show the application name and its memory footprint. The next set of sub-columns capture the behavior with un-fragmented (Table 4.3) and fragmented physical memory (Table 4.4). Physical memory is said to be fragmented if the free memory is scattered in small holes i.e., non-contiguous. Typically, physical memory is un-fragmented only if the system is freshly booted and/or there is little memory usage. But, the memory gets quickly fragmented as applications/OS allocate and de-allocate memory.

Table 4.3 shows that in the absence of fragmentation, the page fault handler alone (page-fault only) can map a large portion of application's memory with 1GB pages for three out of eight applications (`XSBench, GUPS, Graph500`). If an application pre-allocates memory in large chunks, then the fault handler would often find the faulting address to be in a 1GB-mappable region, and use 1GB pages. However, `Redis` and `Memcached` incrementally allocate memory while inserting key-value pairs. Thus, the fault handler could map a small portion of its memory with 1GB pages. `SVM, Btree, Canneal` also, do not pre-allocate their entire memory needs.

The behavior is different if physical memory is fragmented. Even if the fault handler finds a 1GB-mappable address range, it is unlikely to find a free 1GB physical memory chunk. Thus, it often falls back to smaller pages. This is evident from the "page-fault only" sub-columns in Table 4.4 as only a few 1GB pages are allocated.

4.4.1.3 Large page promotion

If an application does not pre-allocate memory or when physical memory is fragmented, it becomes important to later re-map (promote) address ranges to larger pages, when possible. Trident extends THP's khugepaged thread to promote to both 1GB and 2MB pages.

Figure 4.5 shows a flowchart of Trident's page promotion algorithm (changes to THP are shaded). khugepaged first selects a candidate process for page promotion and sequentially scans its virtual address space. During scanning, Trident looks for 1GB-mappable virtual address ranges that are mapped with smaller pages. Subsequently, it looks for 2MB-mappable regions mapped with 4KB pages. If a candidate 1GB-mappable range is found, khugepaged requests the buddy allocator for a free 1GB physical memory chunk. If a 1GB chunk is unavailable, khugepaged requests compaction of the physical memory to create one. Trident extends THP's compaction functionality to create a 1GB physical memory chunk (will be detailed shortly). If

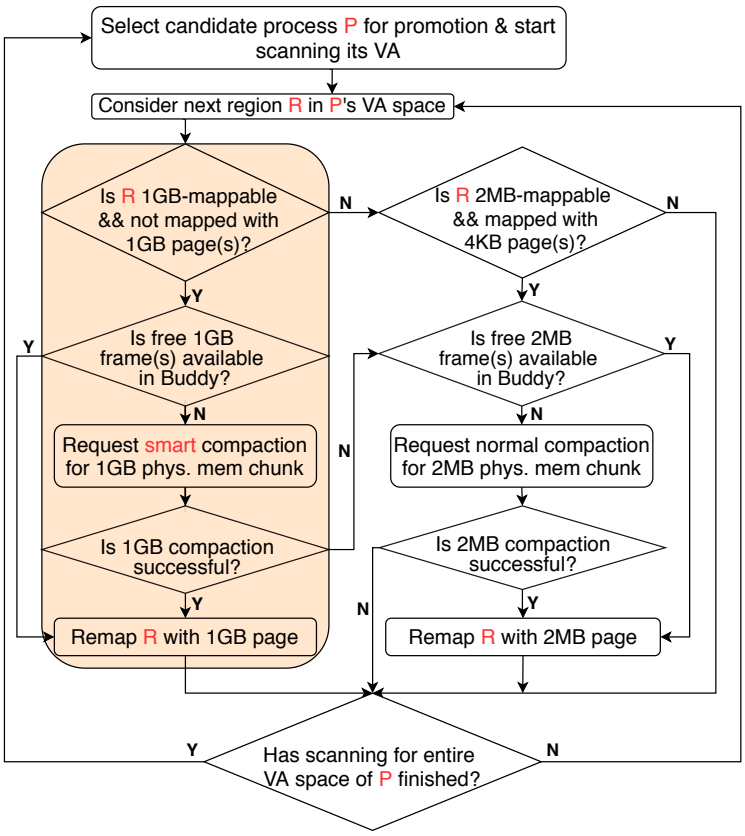


Figure 4.5: Trident's large-page promotion algorithm.

the compaction fails, it attempts to map it with 2MB pages (if not already mapped with 2MB). Trident's policy of preferring 1GB pages but falling back to 2MB pages makes the most out of TLB resources.

Table 4.3 shows the number of 1GB and 2MB pages allocated when the above-mentioned promotion policy is applied along with the page fault handler under unfragmented physical memory. For example, in the un-fragmented case, khugepaged is able to promote about 39GB of memory using 1GB pages for `Redis`, when the fault handler alone failed to allocate even a single 1GB page. `SVM, Canneal` also enjoyed many more 1GB pages due to page promotion.

When the physical memory is fragmented, page promotion helps applications get some 1GB pages, although slightly smaller in number compared to the un-fragmented case (see Table 4.4). For example, 1GB pages allocated to `SVM` drop from 65 to 53. This is expected; free 1GB memory chunks are scarce even after compaction.

Overheads of compaction for 1GB, however, can negate benefits of 1GB pages. Creating even a single 1GB chunk often requires significant memory copying. Copying data creates contention in memory controllers and pollutes caches. It also requires scanning large portions

	Memory footprint (GB)	Unfragmented physical memory (all data in GB)					
		page-fault only		promotion with normal compaction		promotion with smart compaction	
		1GB	2MB	1GB	2MB	1GB	2MB
XSBench	117	114	2.94	116	1.2	116	1.2
GUPS	32	31	1	31	1	31	1
SVM	68.5	54	14.3	65	3.5	65	3.5
Redis	44	0	0.5	39	3.4	39	3.4
Btree	25	0	16.7	16	5.8	16	5.8
Graph500	63.5	59	4.01	60	3.35	60	3.35
Memcached	137	16	121	121	16	121	16
Canneal	32	8	1	30	2	30	2

Table 4.3: Comparison of 1GB and 2MB pages allocated via different mechanisms employed in Trident (without physical memory fragmentation).

	Memory footprint (GB)	Fragmented physical memory (all data in GB)					
		page-fault only		promotion with normal compaction		promotion with smart compaction	
		1GB	2MB	1GB	2MB	1GB	2MB
XSBench	117	6	5.3	79	38.1	80	37.1
GUPS	32	9	2.5	31	1	31	1
SVM	68.5	6	5	53	12.2	54	9.9
Redis	44	0	0	25	10.3	28	14.3
Btree	25	0	11.7	8	12.73	12	8.91
Graph500	63.5	5	5.8	37	24.2	38	23.6
Memcached	137	9	60	12	55	16	60
Canneal	32	6	1	6	21	8	22

Table 4.4: Comparison of 1GB and 2MB pages allocated via different mechanisms employed in Trident (with physical memory fragmentation).

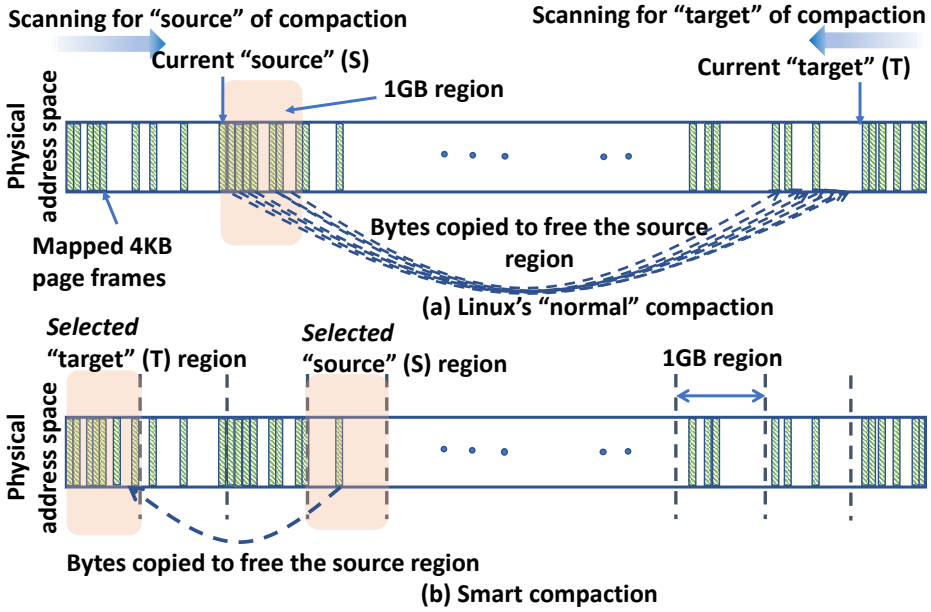


Figure 4.6: Linux's normal compaction (top) and Trident's smart-compaction (bottom).

of physical memory. The application threads could get a smaller fraction of CPU cycles as they can contend with kernel threads performing compaction. In short, it is necessary to reduce the cost of compaction for 1GB pages.

4.4.2 Smart compaction

We, thus, propose a new compaction technique, called smart-compaction, to reduce the cost of 1GB compaction while creating enough 1GB physical memory chunks. The primary goal is to reduce the number of bytes copied. This directly reduces the cost of compaction.

Figure 4.6 illustrates the difference between normal compaction as employed in Linux today and the smart-compaction employed in Trident. Figure 4.6(a) shows the working of normal compaction. On a compaction request, the khugepaged thread starts sequentially scanning physical memory from where it left last time it attempted to compact (remembered in "source" pointer). Scanning starts from the low to high physical addresses. As it finds an occupied physical page frame (4KB) it copies its contents to a free page frame found by scanning in the opposite direction from the "target" pointer. This continues until a free memory chunk of the desired size (e.g., 2MB) is created, or the entire memory is scanned without success.

We observe that this strategy is agnostic to how full or empty a physical memory region is. Consequently, this leads to unnecessary copying overheads. Let us consider the example in Figure 4.6(a). The 1GB region starting at address `S` is mostly occupied and has only 256 free 4KB page frames. Thus, to free that 1GB region, Linux would require copying 999MB of

data (512 × 512 - 256 4KB pages). Instead, if a mostly free region was freed, then the number of bytes copied would be much smaller. While such sub-optimal compaction could be fine for 2MB, it is not so for 1GB, as data copying increases with the page size.

Moreover, if the scan encounters a page frame with unmovable contents (e.g., inodes, DMA buffers), then all copying so far for a region, is wasted (we discussed the effect of unmovable pages on fragmentation in detail in Chapter 2). A free chunk cannot have any unmovable contents. The probability of encountering unmovable contents is much more for a 1GB region.

To address these shortcomings, we propose smart-compaction. The key idea is to divide the physical memory into 1GB regions and select (not scan for) a region with the least number of occupied page frames for freeing (i.e., the source of copying). Similarly, a region with the most number of occupied page frames is preferred as the target for copying. This strategy minimizes data copy. We also track if a given 1GB region contains any unmovable contents. We avoid selecting regions with unmovable content for freeing (i.e, source). This eliminates unnecessary data copying in futile compaction attempts.

To implement the above idea, we first introduced two counters for each 1GB physical memory regions. One counter tracks the number of free page frames, and the other one tracks the number of unmovable pages within a region. Whenever a page is returned to the buddy allocator (i.e., freed), we increment the counter for free frames of the encompassing 1GB region. Further, we decrement the counter for unmovable pages if the freed page frame(s) contained unmovable data. Whenever a page frame(s) is allocated from the buddy allocator the free counter for the encompassing region is decremented. We increment its unmovable page counter if the allocated page frame(s) would contain unmovable data (e.g., requested for allocating kernel data structures). Note that a 1GB region can also have a 2MB page allocated within it. We treat it as 512 base pages for ease of keeping statistics.

As depicted in Figure 4.6, smart-compaction starts by selecting a 1GB region with largest number of free page frames and without any unmovable pages as the source (S). It then selects a target region (T) to move the contents of occupied page frames in the source. The region with the least number of free page frames is selected as the target. It can happen that T may not have enough free frames to accommodate all of S's page frames. If so, a region with next least number of free frames is selected to accommodate the remaining pages (and, so on).

The sub-columns for smart-compaction in Table 4.3 and Table 4.4 show the number of 1GB and 2MB pages that were allocated under un-fragmented and fragmented physical memory, respectively. The number of 1GB pages allocated to each application is the same as that under normal compaction in the un-fragmented case. Under fragmentation, smart-compaction typically provides even more 1GB pages. This is because smart-compaction always selects a

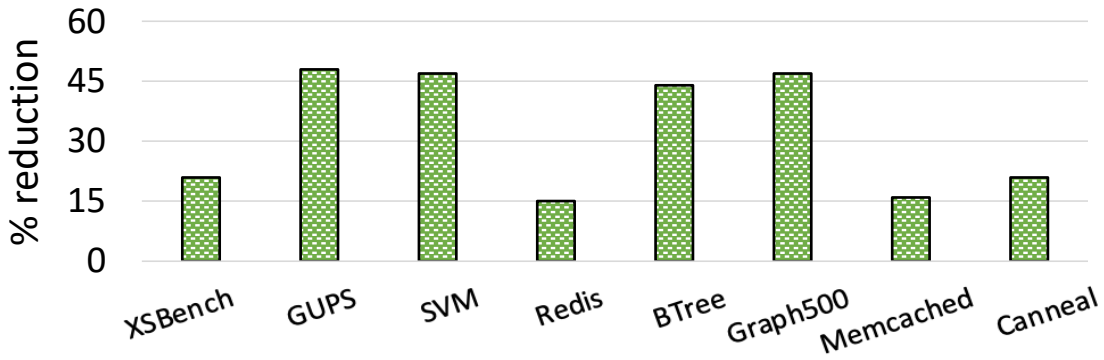


Figure 4.7: Reduction in the number of bytes copied by smart-compaction.

1GB region that is easiest to free, and thus, compaction succeeds more often.

Figure 4.7 shows the percentage reduction in the number of bytes copied with smart-compaction over normal compaction. This measurement is performed when physical memory is fragmented as otherwise compaction is unnecessary. We observe that smart-compaction often reduces the number of bytes copied by up to 85%. This demonstrates that smart-compaction performs less work to create the same or more number of 1GB chunks. Only for `XSBench`, the improvement is less. `XSBench` uses a large fraction of total memory in the system and thus, even the ideal compaction algorithm would not be able to avoid data copy under fragmentation.

4.5 Trident^{pv}: Paravirtualizing Trident

Under virtualization, Trident can be deployed both in the guest OS and in the hypervisor to bring benefits of dynamic allocation of all page sizes, including 1GB pages, to both the levels of address translation. We observe that it is possible to further optimize certain guest OS operations with paravirtualization.

The guest OS copies contents of memory pages to: (1) compact gPAs, and (2) promote address mapping between gVA and gPA to larger pages. While the cost of copying 4KB pages is not high, copying 2MB pages in order to compact or promote them to a 1GB page is slow. We observe that the effect of copying guest physical pages can be mimicked by simply altering the mapping between corresponding gPAs and hPAs. This copy-less approach quickens both compaction and 1GB page promotion in the guest but needs paravirtualization. We call this extension Trident^{pv}.

For brevity, we explain the idea behind Trident^{pv} with the help of large page promotion only (Figure 4.8). Let us assume that two contiguous guest virtual pages, v1 and v2, are currently mapped to two non-contiguous smaller pages g1 and g3 in guest physical memory (Figure 4.8(b)). For simplicity, we assume that a large page is twice the size of a small page.

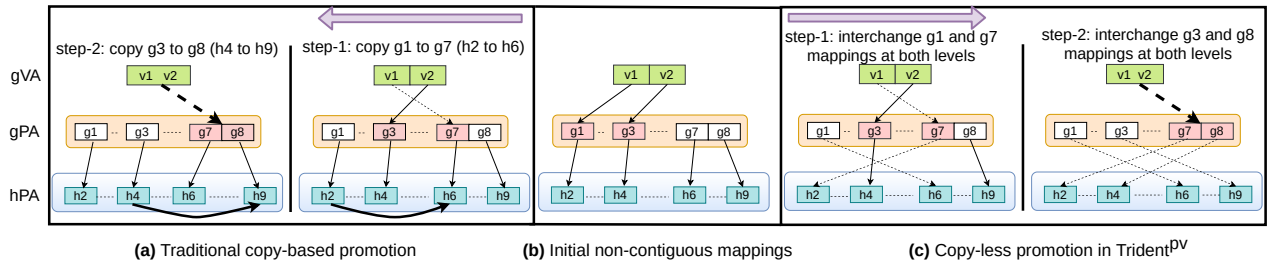


Figure 4.8: Traditional copy-based versus Trident^{pv}'s copy-less page promotion.

To remap gVA encompassing v1 and v2 with a large page, the guest OS first copies their content to two contiguous guest physical pages – g7 and g8. It then updates the mapping between gVA and gPA. This traditional way of promoting large pages by copying contents is shown in Figure 4.8(a).

Figure 4.8(c) shows Trident^{pv}'s approach for page promotion without actual copy. Instead of copying g1 to g7, the hypervisor exchanges the gPA to hPA mappings for g1 and g7. After the exchange, g1 maps to h6 and g7 maps to h2. Since, h2 contains the data originally mapped by g1, this is same as copying g1 to g7. Similarly, the hypervisor exchanges the gPA to hPA mappings for g3 and g8 to create the effect of copying g3 to g8. Later, gVA encompassing v1 and v2 is mapped by the guest with a large page to contiguous gPA encompassing g7 and g8.

The guest OS and the hypervisor need to coordinate for copy-less page promotion and, thus, the need for paravirtualization. Specifically, the guest OS supplies the hypervisor with a list of source and target guest physical pages via a hypercall. The hypervisor then updates the mapping from gPA to hPA in the manner explained above to create the effect of copying guest physical pages. Besides promotion, Trident^{pv} uses the same hypercall for compacting guest physical memory to create 1GB pages in the guest.

While promising, the cost of hypercall ($\approx$ 300ns) to switch between guest and the hypervisors can outweigh the benefits of copy-less promotion. We thus batch requests for multiple page mapping exchanges in a single hypercall. Since a 1GB page is promoted via 512 2MB pages, batch size is known apriori and statically configured. We pre-define two 4KB pages for passing the list of page addresses to exchange between the guest and the hypervisor. One page contains source gPA (here, g1 and g3) and the other contains the target gPAs (here, g7 and g8). In a single hypercall it is thus possible to request exchange for all 512 page addresses. Thus, a single hypercall is sufficient to promote entire 1GB region in gVA mapped with 2MB pages. The hypercall returns after switching all the requested pages or logs any failure in the same shared page used for passing list of pages. On failure, the guest falls back to individually copy contents of pages.

We empirically found that promoting 2MB pages to a 1GB page in the guest takes ≈ 600ms in the copy-based technique. Without batching, Trident^{pv} can promote the same in less than 30ms while batching reduces the time to ≈ 500μs. Note that Trident^{pv}'s copy-less promotion is less useful for promoting 4KB pages to 2MB since the cost of copying 4KB pages is not significant. Hence, we employ copy-less promotion and compaction for 1GB pages only.

4.6 Evaluation

Our evaluation answers the following questions: (1) can Trident improve performance of memory-intensive applications over 2MB transparent huge pages of Linux? (2) how different components of Trident contribute to its performance win over Linux THP? (3) how does Trident perform under virtualization?, and (4) finally, how does Trident^{pv} impact page promotion/compaction in the guest OS?

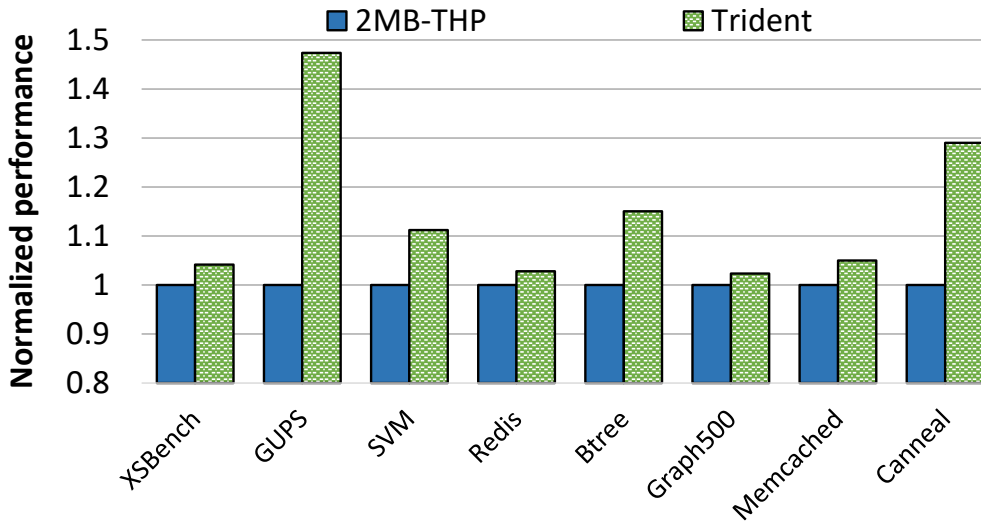
4.6.1 Performance evaluation on bare-metal systems

Performance under un-fragmented physical memory: Figure 4.9a shows the performance of Trident normalized against that of THP (higher is better). For each application, there are two bars in the cluster corresponding to the two configurations we evaluated. Measurements in Figure 4.9a were performed with un-fragmented physical memory.

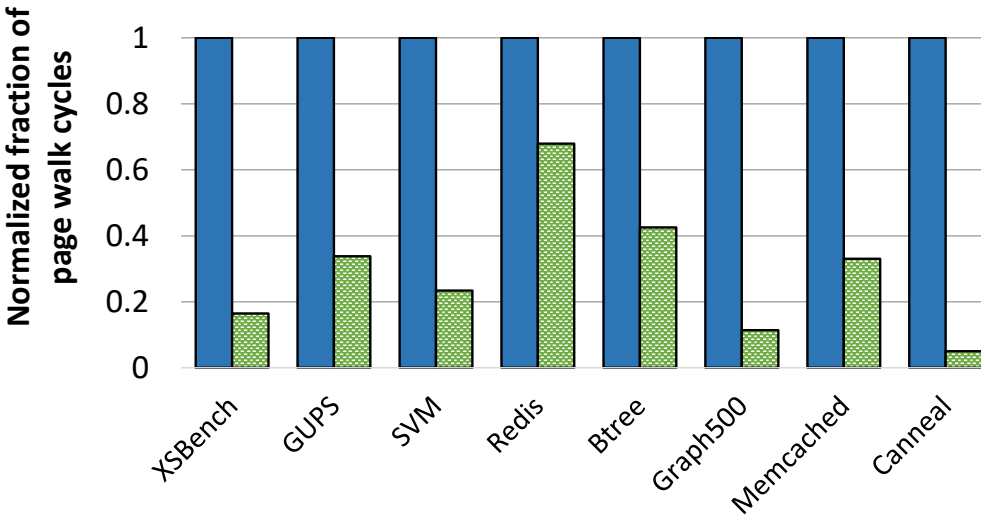
We observe that Trident improves performance over Linux's THP by 14%, on average and up to 47% for `GUPS`. Applications like `XSBench`, `SVM`, `Btree` and `Canneal` witnessed 4.1%, 11.2%, 15% and 30% performance improvement, respectively. Even if we exclude the micro-benchmark `GUPS`, performance improvement is 12%, on average, over THP.

Performance under fragmented physical memory: Arguably, performance analysis under fragmented physical memory paints a more realistic execution scenario. Figure 4.10a shows the normalized performance under fragmented physical memory for the same configurations as before. Trident speeds up applications even more under fragmentation. This is unsurprising since Trident's smart compaction adds a further edge here. On average, it improves performance by 18% over THP and `GUPS` quickens by over 50%. Even excluding `GUPS`, the improvement is 13% over THP.

We also measured how often the fragmented physical memory prevents Trident from mapping an address range with a 1GB page. Table 4.5 shows the percentage of attempts to allocate a 1GB page that fails due to fragmentation. The "NA"s under page fault for `Redis` and `Btree` signify that fault handler never attempts to allocate 1GB pages for them due to lack of 1GB-mappable virtual address ranges during faults. We observe that 71-94% of 1GB page allocations fail due to lack of contiguous physical memory. Even during promotion, 1GB allocations fail



(a) **Normalized performance**



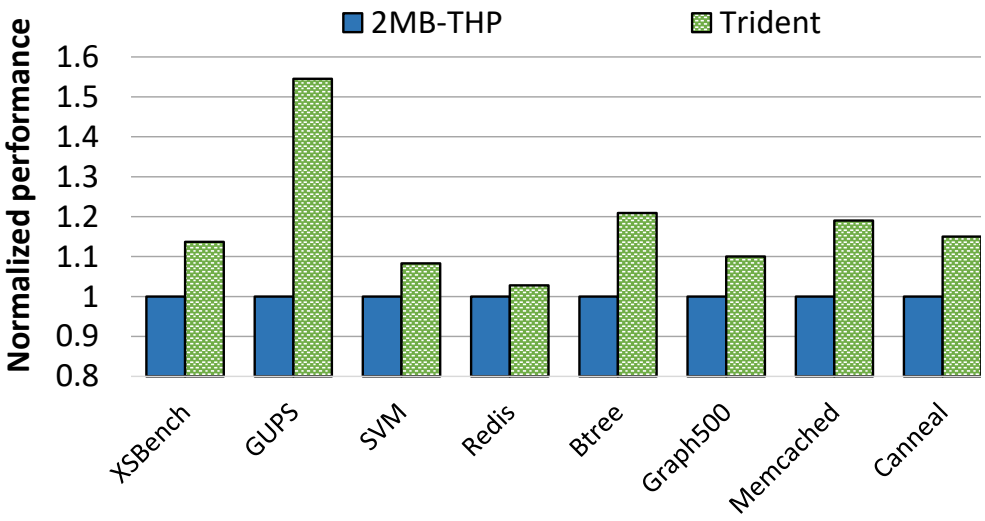
(b) **Fraction of page-walk cycles**

Figure 4.9: Performance under no fragmentation.

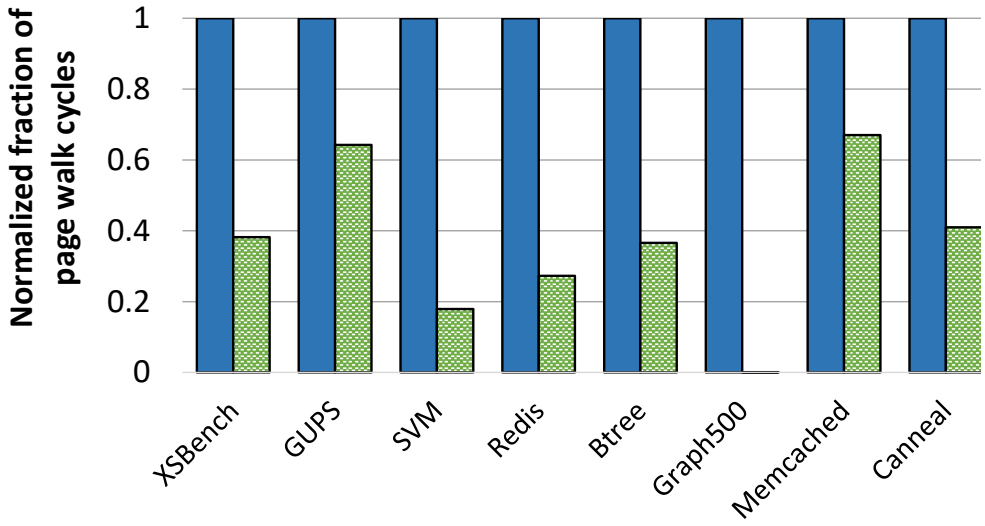
often. This reinforces the need to utilize all large page sizes. Even if the largest page size cannot be used, a smaller large page (2MB) could be deployed.

Impact on page walk cycles: Figure 4.9b and Figure 4.10b show the normalized fraction of walk cycles for THP, and Trident under un-fragmented and fragmented physical memory, respectively. The reductions in the page walk cycles with Trident over THP are significant – 38-85% and 40-97%, under no fragmentation and fragmentation, respectively. Across all configurations, the relative speedups largely correspond to relative reductions in walk cycles.

Impact on tail latency: Tail latency is an important metric for interactive applications (e.g.,



(a) **Normalized performance**



(b) **Fraction of page-walk cycles**

Figure 4.10: Performance under fragmentation.

`Redis`, `Memcached`) that should abide by strict SLAs [121]. Table 4.6 reports 99 percentile latency of `Redis` and `Memcached` under different configurations. Trident does not hurt tail latency relative to both 4KB and THP even though it employs 1GB pages dynamically. Trident reduced TLB misses in the critical path and ensured compaction, promotion and zeroing of 1GB pages happen in background to avoid affecting the tail latency.

	Page fault	Promotion		Page fault	Promotion
XSBench	94%	32%	GUPS	71%	0%
SVM	88%	19%	Redis	NA	36%
Graph500	91%	38%	Btree%	NA	25%
Memcached	43%	81%	Canneal	12%	92%

Table 4.5: Percentage 1GB memory allocation failures

	Redis			Memcached		
	4KB	THP	Trident	4KB	THP	Trident
No-fragmentation	47.3	50.3	46.6	1.53	1.52	1.53
Fragmentation	53.3	53.3	52	1.55	1.55	1.54

Table 4.6: Tail latency (ms) for Redis and Memcached

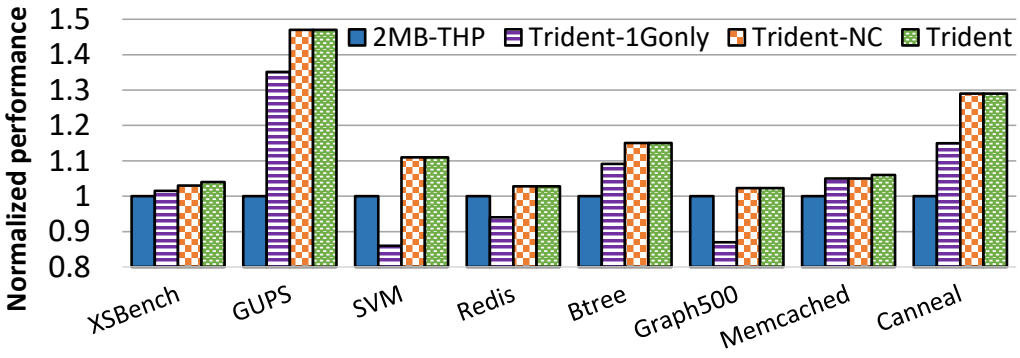
4.6.2 Evaluating Trident's design components

Two of the key aspects of Trident's design philosophy are: (1) the use of all three page sizes, including 1GB pages, and (2) smart-compaction. It is natural to investigate how they contribute in Trident's overall performance.

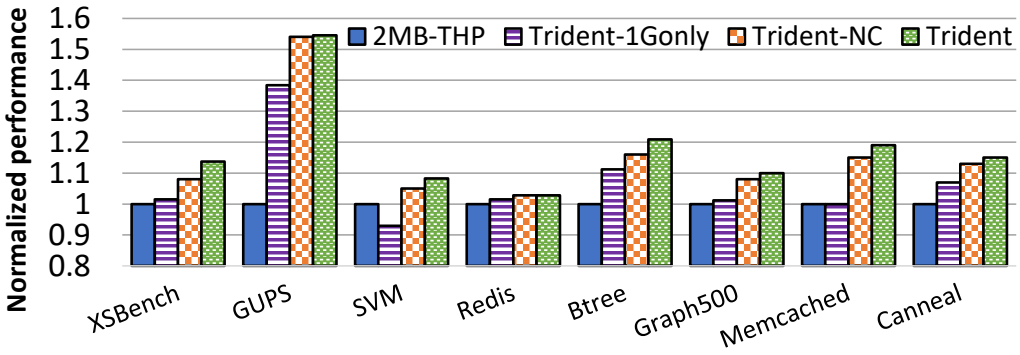
Figure 4.11 teases out the impact of these two factors in Trident's performance, with and without fragmentation (subfigures). Specifically, we introduce two new configurations. Trident-1Gonly denotes the configuration where Trident is disallowed to use 2MB pages. The difference between Trident-1Gonly and Trident highlights the importance of leveraging all large page sizes. Trident-NC denotes the configuration where Trident is allowed to use all three page sizes but barred from employing smart-compaction. Instead, it uses normal compaction available in Linux. The difference between Trident-NC and Trident shows the direct performance implications of smart-compaction, beside reducing data movement. Smart compaction enables Trident to compact fragmented physical memory faster and thus, applications can get 1GB pages sooner in their execution, aiding performance.

First, we focus on the performance when the memory is un-fragmented (e.g., a freshly booted system) in Figure 4.11a. We observe that there is a significant performance gap between Trident-1Gonly and Trident, justifying the need for using all three page sizes. Trident-1Gonly loses performance even relative to THP for several applications (e.g., `Graph500, SVM`). In hindsight, this is expected at this point.

Our analysis in Subsection 4.3.3 revealed that these applications have significant portions of their virtual memory that is 2MB-mappable but not 1GB-mappable. Further, these portions also witness a relatively larger number of TLB misses. Trident-1Gonly is forced to map these



(a) **Normalized performance under no fragmentation**



(b) **Normalized performance under fragmentation**

Figure 4.11: Performance analysis of different components of Trident.

1GB-unmappable regions with 4KB pages and thus experiences higher translation overheads compared to Trident that could deploy 2MB pages. In the process, Trident-1Gonly's benefits from using 1GB pages are more than negated by the overheads of mapping frequently accessed memory with 4KB pages.

Next, we observe no difference in performance between Trident-NC and Trident, i.e., no impact of smart compaction when the memory is un-fragmented (Figure 4.11a). This is expected since compaction is not required when the memory is un-fragmented. Figure 4.11b shows the behavior when the physical memory is fragmented. Here, we see significant performance improvement with smart compaction for several applications. For example, smart compaction alone speeds up `XSBench` by 6%. Similarly, smart compaction is instrumental in improving performances of `SVM`, `Btree`, `Graph500`, `Memcached`, and `Canneal` by 2-5%. Only `GUPS` and `Redis` show no significant performance uplift with smart compaction. In short, the use of all large page sizes and smart compaction both play major roles in Trident's performance.

Comparison with static allocation: Since Trident is a dynamic page allocation technique, we compared it against the alternative dynamic approach i.e., THP. However, one may wonder how Trident compares against static technique of allocating 1GB pages via 1GB-Hugetlbfs.

Unfortunately, 1GB-Hugetlbfs needs prior reservation of contiguous physical memory for 1GB pages. Consequently, it fails when the memory is fragmented, as it often happens in real execution scenarios. Thus, 1GB-Hugetlbfs can be compared only when the physical memory is un-fragmented as in a freshly booted system (performance reported in Figure 4.1b). Here, also, Trident performs 3% better than 1GB-Hugetlbfs, on average, even though Trident does not require user intervention, recompilation or reservation of the physical memory. This was possible since Trident could even map the stack portions using large pages, unlike 1GB-Hugetlbfs, and applications such as `GUPS` and `Redis` are sensitive to TLB misses on the stack region. Only in one application `Btree`, 1GB-Hugetlbfs performs better than Trident. This is because `Btree` allocates virtual memory incrementally over time. Thus, 1GB pages are allocated only during page promotion by Trident and not during page faults upon first access. In contrast, 1GB-Hugetlbfs uses 1GB pages irrespective of virtual memory allocation size at the cost of bloating memory footprint.

4.6.3 Performance under virtualization

We measured the performance of applications running inside a virtual machine with Trident deployed both at the guest OS and at the hypervisor (KVM). Figure 4.12 shows the speedups, normalized to THP deployed in the guest OS and KVM (unfragmented physical memory). Under virtualization, Trident improves performance by 16% on average, over THP. `Canneal` saw biggest improvement (50%), but other applications also benefited significantly. For example, `SVM` and `Graph500` witnessed 6% improvement each.

Performance with Trident^{pv}: When the gPA is fragmented, the guest OS must compact and promote pages using THP's `khugepaged` thread. However, a significant CPU usage in the guest OS could mean wasted vCPU time (cost) for a tenant in the cloud. In fact, Netflix reported how their deployments on Amazon EC2 can get adversely effected by high CPU utilization due to THP's threads [99]. We, therefore, evaluate Trident^{pv} with fragmented gPA but limit `khugepaged`'s CPU utilization in the guest to maximum of 10% of a vCPU. This setup helps to find out whether Trident^{pv}'s faster copy-less promotion/compaction can be useful to use 1GB pages.

Figure 4.13 shows the performance of Trident and Trident^{pv} normalized to THP. Trident^{pv} is more effective than Trident for `XSBench, GUPS, Memcached`, and `SVM` by 5%, on average and by up to 10%. We observe that Trident^{pv} does not always improve performance over

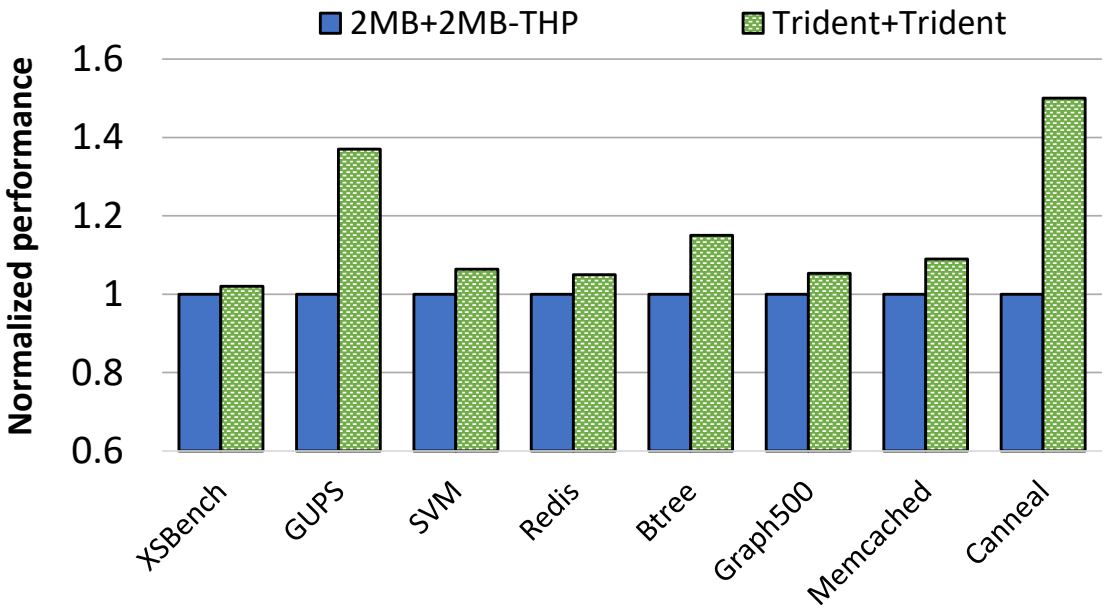


Figure 4.12: Performance under virtualization.

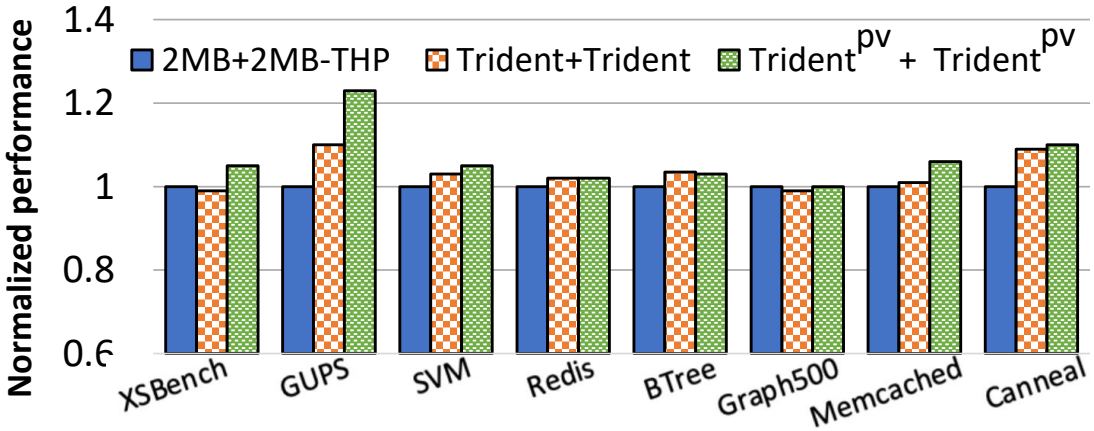


Figure 4.13: Trident^{pv}'s performance under fragmented gPA.

Trident. Recall that Trident^{pv}'s hypercall-based copy-less approach is quicker than the copy-based approach only during promotion/compaction of 2MB pages to 1GB pages. Otherwise, the overhead of the hypercall and that of altering PTEs overshadows the benefits of avoiding copy. In applications such as `BTree, Graph500, Canneal`, 4KB pages are often promoted directly to 1GB pages without needing to go via 2MB pages limiting Trident^{pv}'s scope for improving their performance.

Memory bloat: Large pages are well-known to increase memory footprint (bloat) due to internal fragmentation. Larger the page size more is the bloat. Trident causes bloat in two

out of eight workloads. It adds 38GB and 13GB bloat for `Memcached` and `Btree` over THP. We were able to recover the bloat by simply incorporating HawkEye's technique for dynamic detection and recovery of bloat by demoting large pages and de-deuplicating zero-filled small pages (Section 3.3).

4.7 Summary

While OS support for 2MB pages has matured over the years, 1GB pages have received little attention despite being present in the hardware for a decade. We propose Trident to leverage architectural support of all page sizes available on x86 processors, while also dealing with the latency and fragmentation challenges. Our evaluation shows that 1GB pages, in tandem with 2MB pages, significantly speed up several applications. Further, the paravirtualized extension of Trident, called Trident^{pv}, can effectively virtualize 1GB pages with copy-less page promotion and compaction. Trident's 18% performance gain over Linux THP is likely to motivate researchers to further explore the role of large pages, beyond 2MB.

Chapter 5

Mitigating NUMA Effect on Address Translation

So far in this dissertation, we have discussed how large memory workloads are subject to high virtual memory overheads. The primary source of this overhead is frequent page table walks that happen is response to TLB misses. Since page tables reside in memory, a page table walk may add 100s of CPU cycles in the critical path of execution.

In today's systems, physical memory is not only increasing in size, it is also becoming heterogeneous. This induces non-uniform memory accesses in a system whereby some memory can be accessed much faster than the rest. For example, in multi-socket non-uniform memory access (NUMA) architectures, physical memory is distributed across multiple CPUs in a way that each CPU can access its local memory much faster than that of a remote socket. In this chapter of the dissertation, we discuss how NUMA affects address translation, and propose a system called vMitosis to address the associated challenges.

5.1 Introduction

Applications suffer non-uniform memory access latency on modern multi-tier memory systems. As computer systems embrace even more heterogeneity in the memory subsystem, with innovations in die-stacked DRAM, high bandwidth memory, more socket counts and multi-chip module-based designs, the speed differences between local and remote memory continue to grow and become more complex to reason about [137, 185]. Carefully placing, replicating, and migrating data among memory devices with variable latency and bandwidth is of paramount importance to the success of these technologies, and much work remains to be done on these topics.

However, while there is at least prior work on data placement for application pages [40, 66, 78, 111, 157], kernel data structures have been largely ignored from this discussion, primarily due to their small memory footprint. Consequently, most kernel objects are pinned and unmovable in typical OS designs [143, 144, 96] (as discussed in detail in Chapter 2). We argue that the access latency of kernel objects is gaining importance. This chapter focuses on one critical kernel data structure, *the page table*, and shows that virtualized NUMA servers must carefully reason about its placement to enable high performance.

Why page table access latency matters for applications? Page tables are vital to overall system performance. First, big-memory workloads require frequent DRAM accesses for page table walks due to high TLB miss rates [42, 92, 61]. As system memory capacities grow to meet ever-increasing workload data demand, page tables grow proportionally outstripping the coverage of hardware TLBs. Larger address spaces require additional levels in page tables (e.g., Intel's 5-level page tables). TLB misses under virtualization are already expensive – a 2D page table walk over guest page tables (henceforth gPT) and extended page tables (henceforth ePT) requires up to 24 memory accesses that will increase to 35 with 5-level page tables. Finally, a page table walk does not benefit from memory-level parallelism as it is an inherently serial process. Therefore, each long DRAM access adds latency to address translation [61].

In the earlier chapters, we have seen that on a single CPU server, frequent page table walks can add 10-50% execution overhead on important applications. Similar measurement are also commonly reported in the literature [42, 92, 121]. In this chapter, we show that these overheads are as high as 3.1× on multi-socket machines due to sub-optimal placement of page tables that necessitates high latency remote memory accesses during address translation.

Why page table walks require high latency remote memory accesses? Page table walks require remote memory accesses in two scenarious on virtualized NUMA machines. First, Thin VMs/workloads (i.e., those fitting within a single NUMA socket) are occasionally migrated across NUMA sockets. Current OSs and hypervisors use migration to improve resource utilization and performance isolation. Some real-world examples include VMware vSphere that performs periodic NUMA re-balancing of VMs every two seconds [158], and Linux/KVM that migrates processes to improve load-balancing and performance under memory sharing on NUMA systems [178, 74]. While data pages are often migrated along with the threads, page tables are usually pinned in memory. Consequently, workload/VM migration can make page tables permanently remote.

The second scenario involves Wide workloads/VMs (i.e., those spanning multiple NUMA sockets). Wide workloads experience remote page table walks due to a single copy of the

page table; note that their virtual-to-physical address translations are requested from multiple sockets but each page table entry is local to only one of the sockets.

In a separate work Mitosis, I investigated NUMA effects on address translation for *native* systems [39]. In contrast, in this chapter, we analyze the effect of NUMA on 2D page tables and present vMitosis – a system that extends the design principles of Mitosis to virtualized environments.

vMitosis provides various mechanisms to mitigate NUMA effects on 2D page table walks for both the use-cases discussed above. Our design applies *migration* and *replication* to page tables to ensure that TLB misses are serviced from local memory. While replication and migration are well-known NUMA management techniques, virtualization-specific challenges make their practical realization non-trivial. For instance, a hypervisor may or may not expose the host platform's NUMA topology to a VM. We refer to VMs exposed to the host NUMA topology as NUMA-visible and to VMs not exposed to such information as NUMA-oblivious.

In the NUMA-oblivious configuration, the guest OS is exposed to a flat topology in which all memory and virtual CPUs (vCPUs) are grouped in a single virtual socket. This configuration provides resource management flexibility to cloud service providers as NUMA-oblivious VMs can be freely migrated for maintenance, power management, or better consolidation. Further, CPUs and memory can be added to or removed from NUMA-oblivious VMs dynamically, irrespective of NUMA locality. Most of the VMs on major cloud platforms are available under this configuration [163].

In contrast, NUMA-visible VMs mirror the host NUMA topology in the guest OS. It allows performance-critical services to tune their performance; some important workloads are NUMA-aware by design (e.g., databases [122, 172]), while others leverage OS-level optimizations. NUMA-visible VMs, however, disable hypervisor features such as vCPU hot-plugging, memory ballooning, and VM migration [119, 138]. This is because the current system software stack cannot adjust NUMA topology at runtime. Thus, NUMA-visible VMs limit the resource management capabilities of the hypervisor. However, the choice of a particular configuration is use-case specific, and hence we handle both configurations in our design.

We implement our design in Linux/KVM and evaluate it on a 4-socket NUMA server with 1.5TB physical memory. We show that vMitosis improves performance by migrating and replicating gPT and ePT. The performance improvement is $1.8 - 3.1\times$ for Thin workloads with page table migration, and $1.06 - 1.6\times$ for Wide workloads with page table replication. Our evaluation shows that the page table walks of many applications become less susceptible to the effect of NUMA while using 2MB pages. However, some applications gain up to $1.47\times$ speedup with vMitosis over using 2MB pages.

System	Guest page tables		Extended page tables	
	Migration	Replication	Migration	Replication
Linux/KVM	No	No	No	No
Mitosis	via Replication*	Yes*	No	No
vMitosis	Yes	Yes	Yes	Yes

Table 5.1: NUMA support for page tables in state-of-the-art systems. (*) Replication is possible in Mitosis only if the server's NUMA topology is exposed to the guest OS.

Contributions over Mitosis: Recent proposal Mitosis replicates the page tables on native systems. vMitosis makes several novel contributions over Mitosis (see Table 5.1). First, Mitosis requires the physical server's NUMA topology for replicating the page tables. In bare-metal servers, the OS extracts platform specifications from ACPI tables. However, the hardware abstract layer in virtualized systems often hides platform details from the guest OS. Therefore, Mitosis can replicate gPT only in the NUMA-visible VMs. vMitosis supports both the VM configurations; it re-uses Mitosis in the NUMA-visible VMs but introduces two novel techniques for replicating gPT in the NUMA-oblivious VMs (Subsection 5.3.3).

Second, Mitosis does not provide ePT-level optimizations; vMitosis does. Finally, vMitosis handles page table migration differently from Mitosis. To migrate page tables, Mitosis first replicates them on the destination socket, configures the system to use the new replica, and then releases the old replica. In contrast, vMitosis incrementally migrates page tables when the OS/hypervisor migrates data pages (Subsection 5.3.2). For single-socket workloads, incremental page table migration of vMitosis provides similar address translation performance as the pre-replicated page tables of Mitosis but with lower space and runtime overheads.

5.2 Analysis of 2D page table placement

We start by uncovering the sources of remote DRAM accesses during page table walks and quantifying their impact on performance with a range of memory-intensive applications listed in Table 5.2. We focus on classes of workloads prevalent in data processing or virtual machine deployments that experience high TLB miss rates. Further, a non-negligible fraction of their page table accesses is serviced from DRAM (i.e., miss in the cache hierarchy) due to their random access patterns. Many other big-memory workloads are known to exhibit such characteristics [61].

To simplify the discussion, we partition workloads into two groups to separately demonstrate the two use-cases that lead to remote page table accesses. In the first use-case (referred to as Thin) a workload executes within a single NUMA socket. In the second use-case (referred to

Workload	Description
Memcached	**A multi-threaded in-memory key-value store** [89]
Wide	1280GB dataset, 4B keys, 24B queries 100% reads
Thin	300GB dataset, 20GB slab, 9M queries
XSBench	**Monte Carlo neutron transport compute kernel** [176]
Wide	1375GB input, g=2.8M, p=75M
Thin	330GB input, g=0.68M, p=15M
Canneal	**Simulates routing cost optimization in chip design** [62]
Wide	380GB dataset, # 1200M elements
Thin	64GB dataset, # 240M elements
Graph500	**Generation, search and validation on large graphs** [177]
Wide	1280GB, scale=30, edge=52, 4 iterations
Redis	**Single-threaded in-memory key-value store** [12]
Thin	300GB dataset, 0.6B keys, 100% reads.
GUPS	**Measures the rate of random in-memory updates** [20]
Thin	1 thread, 64 GB input, 1B updates.
BTree	**Measures index lookup performance** [183]
Thin	1 thread, 330GB input, 3.4B keys, 50M lookups.

Table 5.2: Detailed description of the workloads.

as Wide) a scale-out workload spans multiple NUMA sockets using all the CPUs and memory of the system. Our experimental platform is a 4-socket Intel Xeon based Cascade Lake server with 96 cores and 1.5TiB of DRAM (see Section 5.4 for more details).

5.2.1 Analysis of thin workloads

Thin workloads are often migrated across NUMA sockets to reduce power consumption, improve load-balancing, and optimize performance when memory sharing is possible across VMs [178, 74]. We observe that migration of VMs or workloads makes page tables remote. We first describe how the gPT and ePT become remote, and then quantify the performance impact of remote page tables.

Consider, for example, a case where the hypervisor migrates a VM from one NUMA socket to another. In this case, the hypervisor migrates the VM's memory to the new socket with NUMA-aware data migration. During such a migration, the hypervisor also migrates the gPT since gPT pages are like any other guest data pages to a hypervisor. However, hypervisors pin ePT pages in memory and thus the ePT becomes remote after VM migration.

Alternatively, if a NUMA-visible guest OS migrates one of its workloads to another virtual NUMA socket, gPT accesses become remote. This happens because kernel data structures, including page tables, are pinned in typical OS designs today. If ePT were populated by the

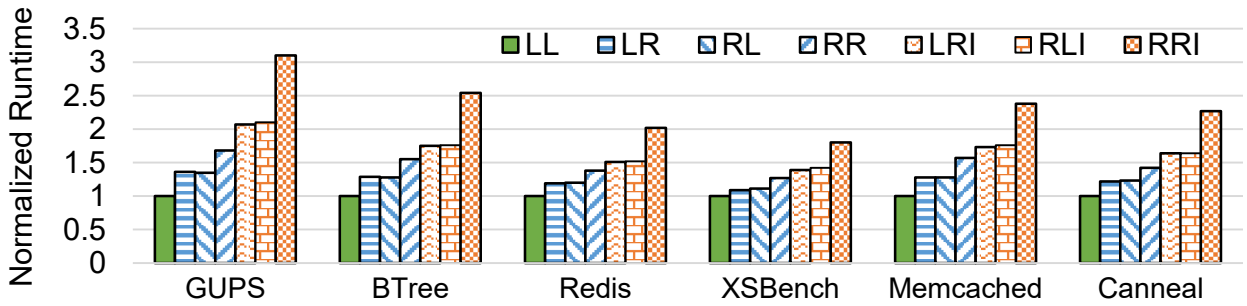


Figure 5.1: Performance impact of gPT and ePT placement configurations on Thin workloads. Details of the configurations are discussed in Table 5.3.

Config	CPU	Data	gPT	ePT	Interference
LL	A	A	A (Local)	A (Local)	None
LR	A	A	A (Local)	B (Remote)	None
RL	A	A	B (Remote)	A (Local)	None
RR	A	A	B (Remote)	B (Remote)	None
LRI	A	A	A (Local)	B (Remote)	B (Remote)
RLI	A	A	B (Remote)	A (Local)	B (Remote)
RRI	A	A	B (Remote)	B (Remote)	B (Remote)

Table 5.3: CPU, data, gPT and ePT placement for different configurations. A and B represent two different sockets in the system (e.g., A=0, B=1). "I" represents interference due to a different workload.

hypervisor on a different NUMA socket prior to workload migration, then ePT also becomes remote post migration. In long-running cloud instances, it is therefore easy to observe that any combination of local/remote gPT and ePT can arise depending on how and when workloads are migrated.

It is important to highlight that ePT may become remote even without migration. ePT pages are allocated in response to virtualized page-faults that are referred to as ePT violations. A vCPU raises an ePT violation when a required translation is absent in the ePT. If fixing a virtualized page-fault requires an ePT page allocation, the hypervisor allocates the page from the local socket of the vCPU that raised the fault. However, ePT is shared across all vCPUs of a VM. Thus, it is possible that ePT pages are allocated on one NUMA socket by a guest workload, but the same translations are later re-used by other guest workloads running on other sockets.

We quantify the performance impact of remote page tables with seven different configurations listed in Table 5.3: the first character denotes if gPT is allocated from local (L) or one of the remote (R) sockets while the second character denotes the same for ePT. For these experi-

ments, we modify the guest OS (Linux) and the hypervisor (KVM) to control the placement of gPT and ePT on specific sockets. The workload threads and data pages are always co-located on the same NUMA socket. This allows us to measure the NUMA effects of page table walks in isolation.

Figure 5.1 shows the runtime, normalized to the best-case configuration LL in which all page tables are local. Considering the first four bars for each application, we observe that when one of the levels of page table (LR, RL) is allocated on a remote socket, the runtime of the application increases by $1.1 - 1.4\times$: the impact of remote gPT is almost the same as remote ePT. As expected, performance drop is higher when gPT and ePT are both remote (RR).

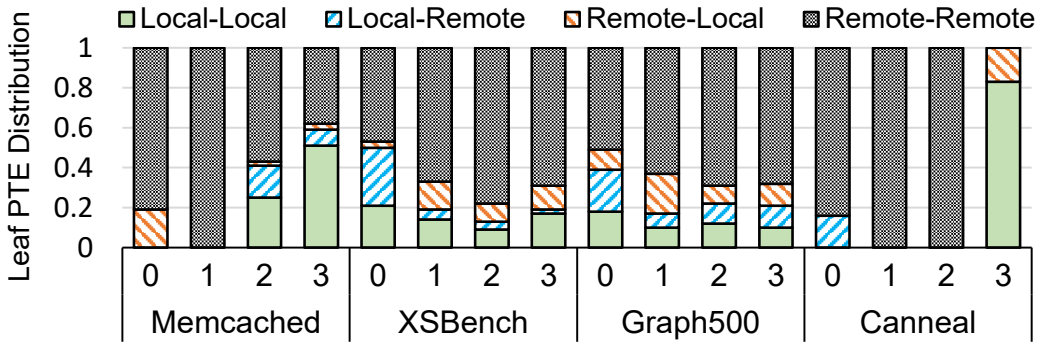
In the experiments so far, we ensured that the remote socket is idle. This enables remote page table accesses to experience uncontended (optimistic) latency. In a real execution scenario, however, the remote socket may be executing other independent application(s). Memory accesses from other processes would thus interfere with remote accesses to page tables under consideration. To measure the impact of contended remote access latency, we add interference by executing `STREAM` micro-benchmark [133] on the remote socket. LRI, RLI, and RRI represent these configurations where ePT, gPT, or both experience contended remote accesses, respectively. As expected, the impact of remote page table accesses is more pronounced under these configurations. In the worst case, remote page tables can cause 1.8 - 3.1× slow down.

5.2.2 Analysis of wide workloads

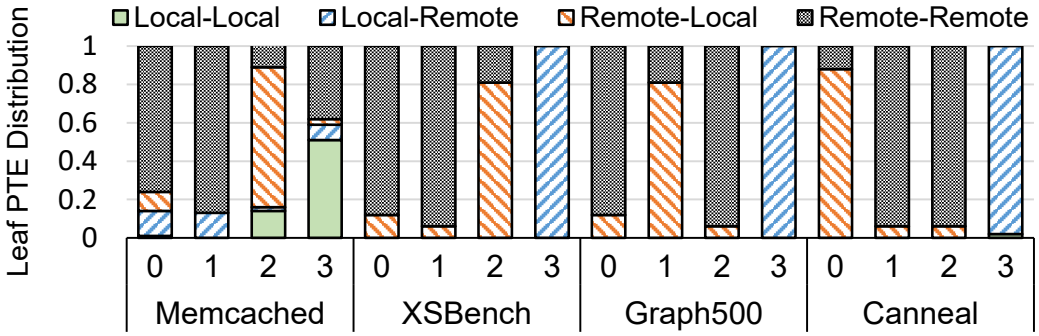
A Wide workload uses resources from two or more NUMA sockets while sharing the same gPT and ePT. For these workloads, each translation entry is remote to all but one socket in the system, irrespective of where their page table pages are placed. A single copy of the page table therefore inevitably leads to remote page table accesses for Wide workloads.

However, unlike Thin workloads, regular data accesses of Wide workloads are interspersed with their page table walks that makes it hard to isolate the impact of remote gPT/ePT accesses. Hence, we adopt a different methodology to estimate the severity of remote page table walks for Wide workloads: we perform an offline 2D page table walk analysis to investigate: (1) what fraction of virtualized page table walks results in one or more remote DRAM accesses and (2) how different configurations of virtualization, i.e., NUMA-visible and NUMA-oblivious, impact page table placement. We discuss the methodology and summarize the observations.

Our analysis focuses on the placement of leaf page table entries (PTEs) since their access latency dominates address translation performance; higher-level PTEs are more amenable to caching by the hardware. We run the workloads to completion and dump the gPT and ePT during their execution periodically once every 5 minutes. We analyze these dumps offline with



(a) **NUMA-visible virtual instance**



(b) **NUMA-oblivious virtual instance**

Figure 5.2: Analysis of 2D page table walk of Wide workloads on NUMA-visible and NUMA-oblivious VMs on a 4-socket machine. Bar for each socket (represented by the number) shows the fraction of 2D page table walks that results in Local-Local, Local-Remote, Remote-Local or Remote-Remote leaf PTE access in gPT and ePT, when TLB misses are serviced for one of the threads running on that socket.

a software 2D page table walker. To estimate the local/remote access ratio of page table walks, we perform address translation for each guest virtual address and record the NUMA socket on which the corresponding leaf PTEs from gPT and ePT are located. Depending on the placement of leaf PTEs, each 2D page table walk is classified into one of the four groups: Local-Local, Local-Remote, Remote-Local and Remote-Remote. The first word denotes if the gPT leaf PTE is local or remote (for a particular socket), and the latter denotes the same for the ePT. We repeat this process on all NUMA sockets to estimate the ratio of local/remote DRAM accesses for 2D page table walks.

Figure 5.2a shows the classification of 2D page table walks in the NUMA-visible configuration. Best translation performance is expected when most page table walks fall into Local-Local group. However, we find that $< 10\%$ page table walks resulted in local memory accesses for both gPT and ePT. Intuitively, this is not surprising. In a system with N NUMA sockets, each PTE (in either ePT or gPT) is local to only one and remote to the other $N - 1$ sockets.

Assuming a uniform distribution of PTEs, the probability of a 2D page table walk resulting in local access in both levels is only $1/N^2$. Hence, on our 4-socket system, we expect only about $1/4^2 \approx 6\%$ page table walks to fall into the Local-Local group. In fact, for a thread running on any socket, out of the 16 possible combinations of leaf PTE placement in gPT and ePT, only one configuration is Local-Local, while nine are Remote-Remote, and three in each Local-Remote and Remote-Local. Thus, more than 50% 2D page table walks result in two remote memory accesses (one for gPT and ePT each) while more than 35% result in one remote access due to either gPT or ePT, in expectation.

Note that there can be exceptions to these observations. For example, `Canneal`'s memory footprint is only 380GB that is slightly above the capacity of a single NUMA socket on our server (350GB). Further, it has a single-threaded memory allocation phase. Hence, almost all memory and page tables were allocated from a single NUMA socket (i.e., Socket-3). In this case, more than 80% of the total 2D page table walks are Local-Local for threads running on Socket-3 (25% of the total threads), while almost all page table walks are Remote-Remote for the rest of the threads. This example also shows that the local (default) memory allocation policy can skew the placement of page tables. Therefore, some threads may experience poorer locality than the others.

Figure 5.2b shows the same analysis for NUMA-oblivious VMs. In this case, Local-Local page table walks are almost non-existent. This is not surprising–the invisibility of NUMA topology in the guest OS leads to an arbitrary placement of gPT pages. Consequently, even for a small workload like Canneal, almost all page table walks involve at least one remote DRAM access. Hence, NUMA-oblivious deployments experience higher address translation overheads.

Summary: In this section, we analyzed how remote page table accesses originate, resulting in up to 3.1× runtime overhead for Thin workloads. Moreover, we showed that a significant fraction of page table accesses is remote for Wide workloads that span multiple NUMA sockets.

5.3 vMitosis: Design and Implementation

Our goal is to ensure that memory accesses in 2D page table walks get serviced from local memory. We achieve this goal by applying two well-known NUMA management techniques – *migration* and *replication*. vMitosis supports three virtual machine configurations: one NUMA-visible (referred to as NV) and two variants of NUMA-oblivious VMs. We refer to the NUMA-oblivious variants as NO-P (para-virtualized) and NO-F (fully-virtualized). Table 5.4 provides a brief overview of NUMA support for page tables under different configurations in current state-of-the-art systems. Table 5.5 provides the same for vMitosis.

	Config.	State-of-the-art
Page Table Migration	NV	gPT: replicate and delete old replica in guest [39] ePT: no migration
	NO-P NO-F	gPT: migrates with data migration in the hypervisor ePT: no migration
Page Table Replication	NV	gPT: replicate in the guest OS [39] ePT: no replication
	NO-P	gPT: no replication ePT: no replication
	NO-F	gPT: no replication ePT: no replication

Table 5.4: Migration and replication of 2D page tables in current state-of-the-art virtualized systems. NV: NUMA-visible, NO-P: NUMA-oblivious-paravirtualized, NO-F: NUMA-oblivious-fully virtualized.

	Config.	vMitosis
Page Table Migration	NV	gPT: migrate incrementally with data migration in the guest OS ePT: allocated to be co-located with data pages in the hypervisor
	NO-P NO-F	gPT: migrates with data migration in the hypervisor ePT: migrate incrementally with data migration in the hypervisor
Page Table Replication	NV	gPT: replicate in the guest OS [39] ePT: replicate in the hypervisor
	NO-P	gPT: replicate in the guest with hypercalls ePT: replicate in the hypervisor
	NO-F	gPT: replicate in the guest OS with data migration in hypervisor ePT: replicate in the hypervisor

Table 5.5: Migration and replication of 2D page tables in vMitosis. NV: NUMA-visible, NO-P: NUMA-oblivious-paravirtualized, NO-F: NUMA-oblivious-fully virtualized.

5.3.1 Design overview

Migration: We propose page table migration for Thin workloads. vMitosis takes a two-fold approach to enable the migration of gPT and ePT. First, we co-locate page tables with data pages. Second, we integrate page table migration with the data page migration policies of the OS/hypervisor. These mechanisms can be enabled independently in each layer.

Replication: We propose page table replication for Wide workloads and VMs. A hypervisor has direct access to the NUMA topology of the system. Therefore, ePT can be replicated by extending the Mitosis design [39]. For replicating gPT, the guest OS needs to: (1) know the number of NUMA sockets the VM is using, (2) allocate gPT replicas on different sockets, and (3) identify the scheduler mapping of vCPUs to NUMA sockets to load each vCPU with its local gPT replica. A NUMA-visible guest OS replicates gPT easily since NUMA topology is exposed

to the guest OS. However, a NUMA-oblivious guest OS requires additional techniques to fulfill these requirements. We propose two different techniques to handle this: the first technique is based on para-virtualization while the second technique is fully-virtualized. We discuss the trade-offs involved in these techniques in Subsection 5.3.3

5.3.2 Page table migration

General design: We start by allocating page tables from the local NUMA socket of the workload (similar to current systems) but additionally use a simple policy to determine when to migrate them. First, we maintain some metadata for each page table page to decide whether it is placed well or needs to be migrated. Note that a page table is a tree of physical address pointers wherein page table entries (PTEs) in the internal levels point to the next-level page table pages while leaf PTEs point to the application data pages. For each page table page, we maintain an array with an entry for each NUMA socket; each array element represents the number of valid PTEs that point to its NUMA socket.

Based on this metadata, we say that a page table page is placed well if it is co-located with most of its children. While we proactively try to allocate page tables close to data, the system software runtime can migrate workloads and data pages at runtime. To account for dynamic scheduling activities in the system, we track page table placement by piggybacking on PTE updates that happen in the page migration path. Since current systems use sophisticated techniques to co-locate data and threads, PTE updates due to data page migration serve timely hints to vMitosis to trigger the migration of page tables as soon as they become remote. We now discuss how gPT and ePT migration works under different modes of virtualization.

5.3.2.1 Page table migration in NV (NUMA-visible) configuration

If a Thin workload is running in a NUMA-visible VM, the guest OS's NUMA-aware scheduler may move the workload from one socket to another. As a result, both the gPT and the ePT may get misplaced and remain remote for the rest of the workload life cycle starting from the point of migration. For NUMA-visible VMs, we expect that the guest OS employs automatic NUMA balancing to co-locate the data and threads of its workloads. In this case, the guest OS will migrate data pages to the new socket but not the gPT.

We leverage the fact that leaf PTEs in gPT get updated when the guest OS migrates application data pages. vMitosis tracks these migrations and updates the counter values in the corresponding page table pages. As soon as most of the PTEs in a leaf gPT page point to a remote socket, vMitosis notices the misplacement of the page and migrates it. Hence, incremental data migration in vMitosis automatically triggers the migration of leaf gPT pages

first. Further, the migration of leaf gPT pages results in updated counter values for the internal (higher) level gPT pages that in turn triggers their migration. This way, page table migration is automatically propagated from the leaf level to the root of the gPT tree.

ePT migration works similarly in the hypervisor. However, optimizing ePT placement requires an additional consideration. Note that a single vCPU may allocate the entire memory for its VM, e.g., when the VM boots with pre-allocated memory or a single guest thread initializes all the memory. Similar to current hypervisors, vMitosis allocates ePT pages on the local socket of the vCPU that requests memory. For a Wide VM, all ePT pages may therefore get consolidated on a single socket while data pages are distributed across multiple sockets. In these cases, the guest OS can migrate data pages at runtime to improve memory access locality. However, NUMA migrations of the guest OS may not be visible to the hypervisor. For example, a guest OS's data page migration does not trigger an ePT violation if the ePT entries of both the old and new data pages are already allocated. The invisibility of guest NUMA migrations can therefore lead to misplaced ePT. To handle such cases, we occasionally invoke automatic page table migration to verify the co-location invariant and migrate misplaced ePT pages.

5.3.2.2 Page table migration in NUMA-oblivious NO-P and NO-F configurations

Under NUMA-oblivious deployments, we expect the hypervisor to co-locate the threads and data pages of guest applications. Note that when the hypervisor migrates guest data pages, gPT is automatically migrated since a gPT page is like a regular VM data page for the hypervisor. Therefore, we do not need to consider a separate gPT migration mechanism for NUMA-oblivious VMs. However, ePT becomes remote if the hypervisor migrates guest applications or the entire VM. We use the same technique here as discussed above: migration of guest physical pages trigger ePT migration in vMitosis from the leaf level to the top of the ePT tree. This way vMitosis achieves local page walks for both ePT and gPT in all configurations.

5.3.2.3 Linux/KVM implementation

We implement ePT and gPT migration in Linux/KVM as an extension to the pre-existing automatic NUMA balancing technique called AutoNUMA [73]. AutoNUMA periodically invalidates PTEs in a process's page table to induce minor page faults. These faults act as a hint for the OS to assess whether a remote socket dominates memory accesses for a data page. In addition to allocating page tables from the local socket during workload initialization, we rely on AutoNUMA-driven data page migration to drive the migration of page tables.

In our implementation, we avoid interfering with regular page table updates by implementing page table migration as another pass on top of AutoNUMA. To do so, we first wait for AutoNUMA to complete fixing the placement of data pages in a specific virtual address space

range, and then scan the corresponding page tables to update the counters and migrate the page tables if necessary. This allows vMitosis to benefit from AutoNUMA's dynamic rate limiting heuristics that adjust the frequency of scanning based on the rate of data page migration. In the normal case where no page table migration is needed, we rarely scan page tables causing no interference in the common case.

To ensure correctness while migrating a gPT page, we acquire a write lock on the per-process `mmap_sem` semaphore. This avoids consistency issues in the presence of split page table locks in Linux where each page table page can be locked independently by different threads [80]. We acquire and release the lock for each gPT page migration separately to avoid latency issues. However, we do not expect this to be a performance issue as page table migration is an infrequent operation and migrating a page table page takes only a few microseconds. In KVM, all ePT updates are already protected by a per-VM spin lock. Therefore, vMitosis does not require additional synchronization for ePT migration.

5.3.3 Page table replication

General design: We enable local address translation for Wide workloads by replicating their page tables. Recently proposed Mitosis replicates page tables for native execution on NUMA machines. However, two levels of the page tables and the hardware abstraction layer of the hypervisor make it non-trivial to extend Mitosis to virtualized environments. This subsection introduces our replication support in vMitosis which builds on the Mitosis design while highlighting the subtle differences and challenges involved in extending such a design.

We extend Mitosis to replicate ePT in both NUMA-visible and NUMA-oblivious configurations. In the NUMA-visible configuration, we also re-use Mitosis to replicate gPT. However, the gPT replication technique of Mitosis is insufficient for NUMA-oblivious VMs since the guest OS has no visibility into the NUMA topology. We propose two new techniques to replicate gPT for such VMs. The first technique replicates gPT via para-virtualization. In this approach, the guest OS relies on the hypervisor to identify NUMA topology, vCPU to NUMA socket mapping, and allocate gPT replicas. The second technique is fully-virtualized wherein the guest OS replicates gPT by discovering the NUMA topology and vCPU scheduling information with a micro-benchmark. Additionally, it leverages the commonly-used *"local"* memory allocation policy of the hypervisor [175] to allocate gPT replicas from different sockets.

5.3.3.1 ePT Replication

The design and implementation of ePT replication is common across all VM configurations. We introduce the following four components in the hypervisor for replicating ePT:

1. **Allocating ePT replicas:** We extend the ePT violation handler to allocate replicas on all NUMA sockets eagerly, i.e., each ePT page allocation is followed by the allocation of its replicas. To allocate ePT replicas from the desired sockets, we introduce a per-socket "page-cache" that reserves some pages on each socket and uses them to allocate ePT pages. When the free memory pool in a NUMA socket falls below a certain threshold (e.g., 10%), the page-cache reclaims memory from the socket by migrating some data pages to another socket or by swapping them out.

2. **Ensuring translation coherence:** The updates to ePT are managed solely by the hypervisor. ePT updates occur when a VM allocates a new data page or due to various hypervisor actions like page sharing, live migration, working set detection, etc. These updates are performed by the hypervisor on the ePT. We eagerly update all replicas when an ePT entry is modified, followed by a TLB flush to ensure translation coherence for the entire VM.

3. **Assigning local ePT replica:** Each virtual CPU (vCPU) of a VM is managed by the KVM hypervisor as a user-level thread that can be scheduled on any physical CPU (pCPU). When a vCPU is scheduled, we provide it with the local ePT replica to ensure that ePT page table walks are performed entirely within its local NUMA socket.

4. **Preserving the semantics of access and dirty bits:** ePT is referenced by the hardware on a TLB miss. Recent architectures have also introduced access and dirty bits on ePT [107]. The hardware page table walker sets these bits when a physical page is accessed or modified; the hypervisor is not involved in their updates. For these bits, ePT replicas may be inconsistent since a hardware page table walker will set them only on its local replica. However, this inconsistency does not compromise correctness. Hypervisors use these bits in various contexts e.g., to decide whether a page needs to be flushed before it can be released. To ensure correctness, we `OR` the value of these bits on all replicas when the hypervisor queries them; the return value is the same as it would be if all replicas were always consistent. Similarly, if the hypervisor clears the access or dirty bits, we reset them on all the replicas.

With these four components, derived from the native Mitosis design and adapted to virtualized systems, we enable ePT replication for virtual machines. Next, we discuss how gPT is replicated under different virtualization scenarios.

5.3.3.2 gPT replication in NV (NUMA-visible) configuration

This is the simplest case for replicating gPT since the physical topology is exposed to the guest OS. We modify the guest OS to reserve the page cache, allocate gPT replicas on different sockets

and program each thread's page table base register with its local replica. We leverage the open source version of Mitosis [154] to replicate gPT in this configuration.

5.3.3.3 gPT replication in NO-P (NUMA-oblivious paravirtualized) configuration

In this configuration, the NUMA topology is not visible inside the VM. The guest OS requires two main pieces of information to replicate gPT: (1) how many sockets are being used by the VM and (2) how vCPUs are scheduled on these sockets. This information is required to identify how many replicas the guest OS should allocate and configure each vCPU with its local replica. We use para-virtualization to resolve both these challenges where the guest OS relies on explicit hypervisor support as discussed below:

1. **Identifying NUMA socket ID of each vCPU:** The guest OS queries the physical socket ID from the hypervisor for all its vCPUs. This serves two purposes: (1) it enables the guest OS to know how many physical sockets it is using. This way, the guest OS knows how many gPT replicas it needs to allocate. (2) it allows the guest OS to schedule its CPU cores with the local gPT replica.

2. **Allocating gPT page caches on specific sockets:** The guest OS populates a per-socket "page cache" based on the number of sockets that the VM is currently using on the host. To ensure local allocation of each page cache on the physical server, the guest OS requests the hypervisor to *pin* these page cache pages onto their intended sockets.

This design allows the hypervisor to perform NUMA-aware scheduling and change the vCPU to pCPU mapping, as per the requirement of dynamic optimizations. To adapt to the hypervisor-level scheduling changes, the guest OS queries the vCPU to socket ID mapping at regular intervals and updates the vCPU to gPT replica mapping as required.

5.3.3.4 gPT replication in NO-F (NUMA-oblivious fully virtualized) configuration

The goal is to replicate gPT entirely within the guest OS without support from the hypervisor and para-virtualization. We achieve this by exploiting two common properties of NUMA systems: (1) two hardware threads from different sockets exhibit higher communication latency compared to threads within the same socket [65, 110]. The is because cross NUMA socket communication is subject to the interconnect latency between different CPUs, and (2) OS/hypervisors commonly use "local" memory allocation policy wherein memory is preferably allocated from the same NUMA socket where the requesting application thread is running [175]. This helps in minimizing high latency remote memory accesses for data pages.

	0	1	2	3	4	5	6	7	8	9	10	11
0	-	125	125	126	**50**	125	126	126	**55**	125	125	126
1	-	-	125	126	126	**50**	125	125	125	**52**	126	125
2	-	-	-	126	125	126	**62**	126	125	125	**55**	125
3	-	-	-	-	125	125	126	**50**	125	126	125	**55**
4	-	-	-	-	-	125	125	126	**62**	126	125	125
5	-	-	-	-	-	-	125	126	125	**55**	125	126
6	-	-	-	-	-	-	-	126	126	126	**50**	125
7	-	-	-	-	-	-	-	-	125	126	125	**52**

Table 5.6: Time to transfer a cache line (in ns) between different vCPU pairs. Bold underlined entries represent vCPU pairs wherein both vCPUs are scheduled on the same NUMA socket. The table is shown partially from the 192x192 matrix we profiled on our system.

We exploit the first property to construct virtual NUMA groups within the guest OS using a micro-benchmark that measures the pair-wise cache-line transfer latency between all vCPUs. Based on these measurements, vMitosis assigns vCPUs to virtual NUMA groups such that the communication latency is low between any two vCPUs in the same group and high for any two vCPUs from different groups.

For example, consider Table 5.6 that shows the cost of transferring a cache line between different vCPU pairs on our experimental platform. Given this cost metric, vMitosis forms four groups of vCPUs (0,4,8), (1,5,9), (2,6,10), and (3,7,11) where each tuple represents a virtual NUMA group. These virtual groups have a one-to-one correspondence with our physical server topology, and hence, we identify the affinity groups of the vCPUs without relying on para-virtualization. In general, we find that the virtual NUMA groups constructed by our micro-benchmark always mirror the host topology, even under interference from other VMs and workloads.

We next leverage the hypervisor's local memory allocation policy to allocate gPT replicas from the local physical socket of each virtual NUMA group. For this, we select one vCPU from each group in the guest to allocate memory for its page-cache immediately upon boot. The vCPU allocates and accesses its page-cache to enforce page allocation in the hypervisor via ePT violations. From this point, each virtual NUMA group references its replica, and gPT replication works as discussed before. When a gPT page is released, we add it back to its original page-cache pool.

In this approach, it is possible that a replica page could not be allocated locally e.g., due to unavailability of free memory on the local socket. For these cases, we expect the hypervisor's NUMA-balancing technique to migrate misplaced replica pages to their expected sockets. Note that different NUMA groups reference different copies of gPT replicas. Therefore, all accesses

for each replica page originate from the same socket. It makes it easier for the hypervisor to identify which of the gPT replica pages are misplaced (if any) and migrate them. In our evaluation, we show that the overheads of misplaced gPT replicas are moderate even in the worst case (when all gPT accesses are remote) because most gPT accesses are already remote in existing systems.

Note that replicating gPT via NO-P or NO-F involves a trade-off regarding the ease of deployment and performance guarantees. NO-P guarantees high performance by providing explicit hypervisor support to the guest OS for satisfying all the requirements of gPT replication. However, cross-layer communication makes NO-P harder to deploy. NO-F is easy to deploy but may lead to suboptimal performance in rare cases when non-local replicas get assigned to vCPUs. Our evaluation shows that NO-F and NO-P provide similar performance in the common case (see Subsection 5.4.2.2).

5.3.3.5 Linux/KVM implementation

KVM maintains a descriptor to store the attributes of each ePT page. We use the original ePT pages as the "master" copy and store references to the corresponding replicas within their descriptors. We then intercept writes to the master ePT and propagate them to all the replicas within the same acquisition of a per-VM spin lock to ensure eager consistency. If a vCPU is rescheduled to a different NUMA socket, we invalidate the old ePT for the vCPU and assign a new replica based on its new socket ID.

We replicate gPT using the open source implementation of Mitosis in the NV configuration. We also use Mitosis as the core gPT replication engine for NUMA-oblivious NO-P and NO-F configurations but augment it with two different guest modules. In NO-P, the guest kernel module issues hypercalls to the hypervisor to determine the physical socket ID of all its vCPUs and allocate local gPT replica for each vCPU group. Under the NO-F configuration, the guest module builds the necessary virtual NUMA groups with a micro-benchmark and allocates one replica page-cache for each group. These modules periodically update their vCPU to NUMA group mappings to adapt to hypervisor-level scheduling changes.

5.3.4 Deploying vMitosis

vMitosis supports per-process/per-VM migration and replication of page tables for unmodified applications. Users can enable or disable page table migration at runtime (enabled system-wide, by default), while replication requires explicit selection by the user with `numactl` NUMA control policy of Linux.

It is important to highlight that the choice of migration or replication depends on the

classification of a workload as either Thin (migration) or Wide (replication). In this chapter, we have primarily focused on building various "mechanisms" to cover real-world scenarios that lead to remote memory accesses for page table walks. Hence, we used simple heuristics (e.g., number of requested CPUs and memory size) and user inputs (e.g., `numactl`) to classify VMs/processes as Thin or Wide. We leave investigating more sophisticated policies as future work, e.g., based on the cpuset allocation, hardware performance counters etc.

5.4 Evaluation

We evaluate vMitosis on real hardware with a selection of memory-intensive workloads. We conduct page table migration experiments for Thin (Subsection 5.4.1) and page table replication experiments for Wide workloads (Subsection 5.4.2). We further explore the trade-offs between replication and migration (Subsection 5.4.3). In all cases, we exclude workload initialization time from performance measurements.

Evaluation platform: We conduct all measurements on an Intel 4x24x2 Xeon Gold 6252 (Cascade Lake) server with 1.5TiB DDR4 physical memory in total, divided equally among four NUMA sockets. The processor runs at a base frequency of 2.10 GHz with a per-socket 35.75MB L3 cache. It contains a private two-level TLB per core with 64 and 32 L1 entries for 4KB and 2MB sized pages, and a unified L2 TLB with 1536 entries. We enable hyperthreading and disable turboboost.

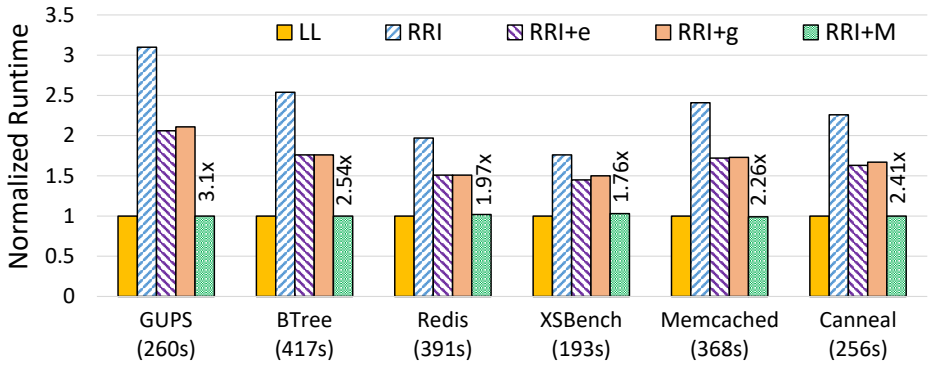
Software configuration: We use Linux v4.17-vMitosis as both the host and the guest OS [154], and KVM as the hypervisor. We pin vCPUs to pCPUs, use `numactl` to select the memory allocation policy, and selectively enable/disable automatic NUMA-balancing and transparent huge pages (THP) depending on the configuration being tested. When enabled, THP is used in both–the guest OS and the hypervisor.

Virtual machines: We configure two VMs using `libvirt` for the KVM hypervisor, each with 192 vCPUs and 1.4TiB of DRAM. NUMA-visible VM divides the DRAM and vCPUs into four virtual sockets with a one-to-one mapping between physical and virtual sockets. NUMA-oblivious VM exports the entire server as a single socket machine.

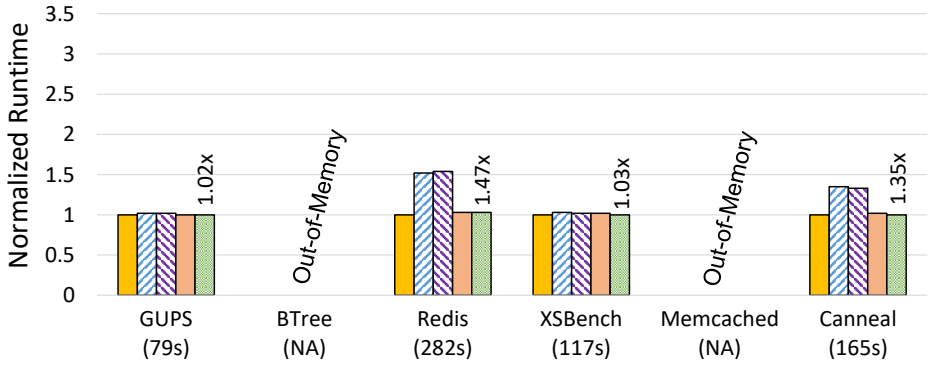
5.4.1 Evaluation with page table migration

This subsection focuses on Thin workloads that fit into one NUMA socket. We show that vMitosis mitigates the effects of remote page table walks when the workload is migrated and scheduled on another NUMA socket.

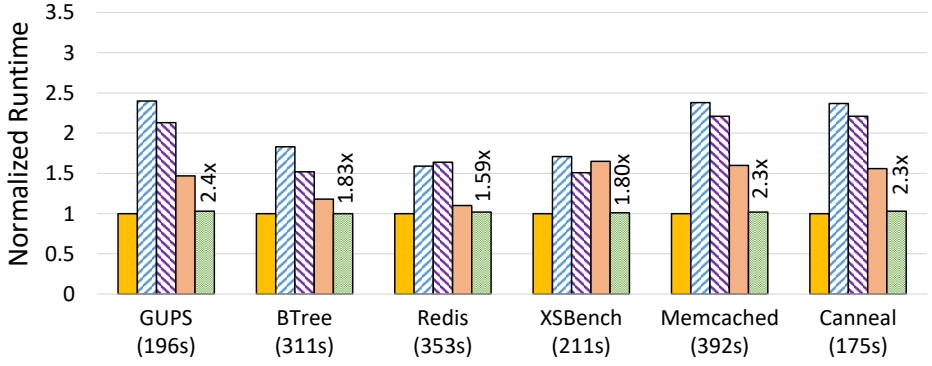
Configuration	gPT	ePT
LL	Local	Local
RRI	Remote+Interference	Remote+Interference
RRI+e	Remote+Interference	Migrated
RRI+g	Migrated	Remote+Interference
RRI+M	Migrated	Migrated



(a) **Using 4KB pages**



(b) **Using 2MB pages**



(c) **Using 2MB pages + fragmentation in guest OS**

Figure 5.3: Workload performance with and without ePT and gPT migration. Bars are normalized to base case (LL). Absolute runtime for the base case in brackets. Numbers at the top show speedup with vMitosis over the worst-case setting (RRI).

Evaluation methodology: We select the NUMA-visible VM configuration to explore various page table configurations for these experiments. In the NUMA-visible case, NUMA controls reside in the guest OS and the hypervisor maintains a 1:1 mapping between the virtual and physical NUMA sockets. We focus on the worst-case situation that occurs when the guest OS migrates one of its workloads; in this case, gPT and ePT both become remote as discussed in Subsection 5.2.1.

We profile the execution for five configurations listed in the table at the top of Figure 5.3. LL represents the best-case performance with local page tables. RRI represents Linux/KVM where ePT and gPT are both remote after workload migration. We measure vMitosis in three configurations; RRI+e replicates only ePT, RRI+g replicates only gPT while RRI+M replicates both ePT and gPT. Additionally, we execute workloads with 4KB and 2MB pages.

Description of results: We show the benchmark results in Figure 5.3. The base case (LL) has both the ePT and gPT on the local NUMA socket. With 4KB pages, all workloads experience performance loss when either ePT or gPT is remote. The worst-case occurs when both are remote, resulting in a slowdown of $1.8 - 3.1\times$. For all six workloads, vMitosis eliminates the overhead of remote page table walks by migrating both levels of the page tables (RRI+M), achieving the same performance as the best case. The effect of ePT or gPT migration is similar as RRI+e and RRI+g contribute roughly half to the overall speedup.

With THP enabled, the difference between the base case and the remote configurations is less visible due to fewer TLB misses with 2MB pages. Therefore, vMitosis provides a relatively modest speedup, except for `Redis` and `Canneal` that gain $1.47\times$ and $1.35\times$ improvement. `Memcached` and `BTree` result in out-of-memory (OOM) due to the well-known internal fragmentation problem with 2MB pages that leads to memory bloat (discussed in Section 5.5).

We also measure performance with THP where the guest OS's physical memory is fragmented. Fragmentation is well-known to limit 2MB page allocations, increasing TLB pressure. To fragment the guest OS's memory, we first warm up the page cache by reading two large files into memory. The total size of these files exceeds memory capacity of the socket where applications execute. We then access random offsets within these files for 20 minutes. This process randomizes the guest OS's LRU-based page reclamation lists. When the application allocates memory, the guest OS invokes its page replacement algorithm to evict inactive pages. Since we accessed files at random offsets, the eviction usually frees up non-contiguous blocks of memory, forcing the allocator to use 4KB pages. However, background services for compacting memory and promoting 4KB pages into 2MB pages remain active.

With THP enabled and fragmented guest OS, vMitosis recovers the performance that was lost due to the lack of 2MB page allocations, resulting in up to $2.4\times$ speedup. `Memcached` and

`BTree` were able to complete their execution in this case since fewer 2MB pages were allocated due to fragmentation. Both these applications also observed significant performance gain with vMitosis. Note that host physical memory is not fragmented and ePT maps guest physical to host physical memory with 2MB pages. Therefore, the speed up here is lower than when both layers use 4KB pages.

Summary: vMitosis effectively mitigates the slowdown caused by remote page table walks by integrating the migration of ePT and gPT with that of data pages. Overall, vMitosis provides up to 3.1× speedup without THP. With THP, we gain up to 2.4× and 1.47× speedup with and without fragmentation.

5.4.2 Evaluation with page table replication

We evaluate the performance benefits of ePT and gPT replication in two settings: NUMA-visible and NUMA-oblivious.

5.4.2.1 Page table replication in a NUMA-visible scenario

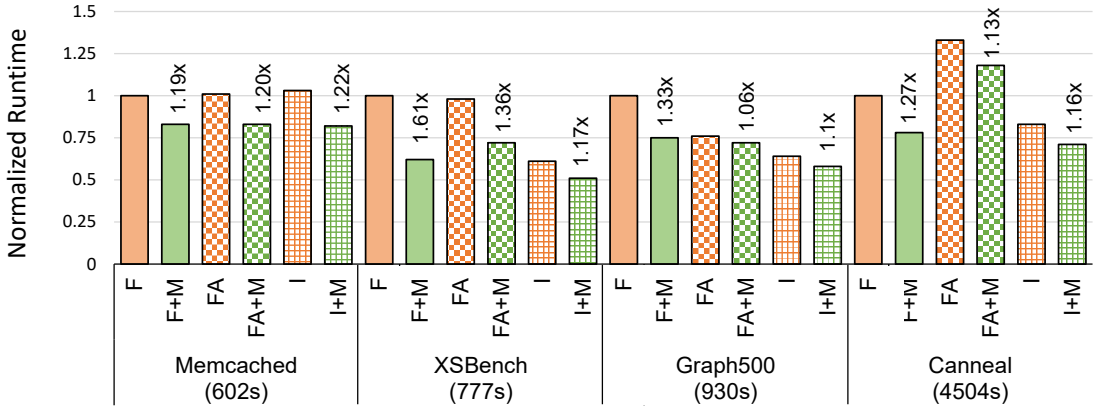
We first measure the speedup due to gPT and ePT replication by giving guest OS access to the NUMA topology.

Evaluation methodology: We set up the guest OS to replicate gPT in our NUMA-visible VM and the hypervisor to replicate ePT. We execute Wide workloads (shown in Table 5.2) inside the VM. We use local memory allocation on the host to match guest memory mappings to the host's NUMA mappings, and pin each vCPU to a pCPU of the respective socket. We use different memory allocation policies in the guest and run each configuration with and without vMitosis. Moreover, we run each workload with and without THP.

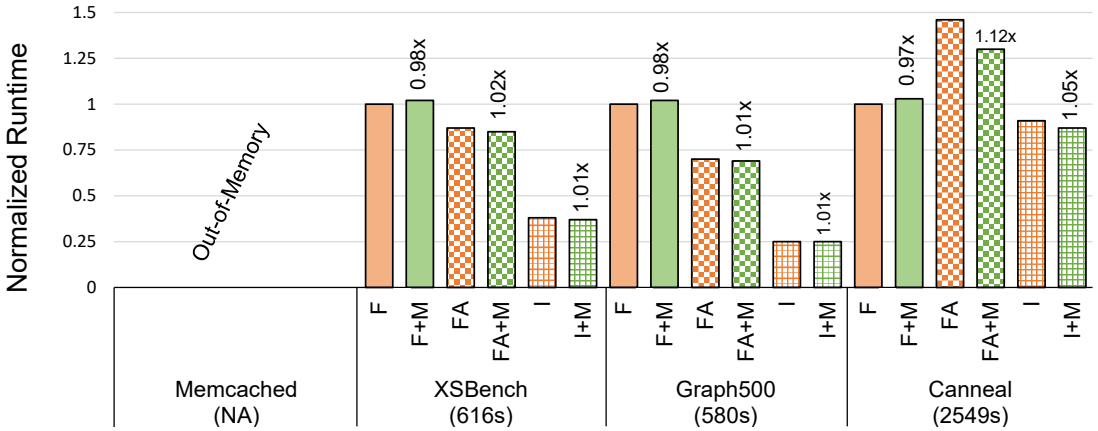
The table on top of Figure 5.4 shows the configurations. F represents workload execution with first-touch (local) memory allocation in the guest OS while FA represents auto page migration enabled on top of first-touch memory allocation. Configuration I represents interleaved memory allocation wherein pages (including gPT and ePT pages) are allocated from all four sockets in round-robin. The vMitosis counterpart for each of these configurations is represented with the suffix +M. For example, configuration F+M represents replicated gPT and ePT, combined with the first-touch page allocation policy for data pages.

Description of results: Figure 5.4a shows the relative runtime of workload execution under six configurations, normalized to the base case (F). We show the absolute runtime (in seconds) for the base case below the workload name. Replicating both gPT and ePT with vMitosis provides $1.06 - 1.6\times$ speedup without any workload changes. Performance improvements due to vMitosis are generally higher in configurations with local memory allocation (i.e., F and

Configuration	Data Allocation	ePT	gPT
F	First-touch	First-touch	First-touch
F+M	First-touch	Replicated	Replicated
FA	First-touch + Balancing	First-touch	First-touch
FA+M	First-touch + Balancing	Replicated	Replicated
I	Interleaved	Interleaved	Interleaved
I+M	Interleaved	Replicated	Replicated



(a) **Using 4KB pages**



(b) **Using 2MB pages**

Figure 5.4: NUMA-visible: Workload performance with and without vMitosis, normalized to the base case (F). Runtime (in seconds) for the base case are in brackets. Numbers at the top show speedup with vMitosis over the corresponding memory allocation policy of Linux/KVM.

FA) – this is because local allocation leads to skewed page table walk traffic in Linux/KVM. Even with a balanced distribution of page tables (configuration I), replicating page tables via vMitosis provides more than 1.10× speedup for all workloads.

We run the same workloads with THP enabled and show the results in Figure 5.4b. All workloads benefit from THP except `Memcached` that resulted in OOM due to memory bloat

created by transparent huge pages. Further, we have found that replicating the ePT alone does not make much of a difference when THP is enabled. Except for `Canneal`, improvements due to vMitosis are negligible. `Canneal` gains 1.12× and 1.05× speedup in first-touch with NUMA balancing and interleaved policies, respectively.

5.4.2.2 Page table replication in a NUMA-oblivious scenario

We next evaluate how vMitosis can improve performance when the NUMA topology is not exposed to the guest OS.

Evaluation methodology: For these experiments, we use Wide workloads in three different configurations as listed in the table on top of Figure 5.5 using the first-touch allocation policy at the hypervisor and pinning vCPUs to pCPUs to ensure stable performance. These settings are consistent with standard virtualization practices [175].

A NUMA-oblivious VM views the system as a single virtual socket. Therefore, only the local memory allocation policy is possible in the guest OS, unlike the NUMA-visible scenario. The base case (OF) represents vanilla Linux/KVM with first-touch memory allocation. We evaluate the two vMitosis variants against the baseline. Configurations OF+M(pv) and OF+M(fv) represent our para-virtualized and fully virtualized solutions, respectively. We enable ePT replication in both the variants.

Description of results: Results are shown in Figure 5.5, normalized to the base case OF with its runtime beneath the workload name. All configurations benefit from vMitosis: ePT and gPT replication provides performance improvements of $1.19 - 1.4\times$ over the baseline using 4KB pages. Enabling THP, we see similar performance characteristics for all configurations. Due to fewer TLB misses and reduced cache footprint of page tables, we see a statistically insignificant speedup of up to 1% with vMitosis.

The performance of both vMitosis variants is roughly similar in all cases. This is an important result highlighting that our fully virtualized approach of replicating gPT entirely within the guest OS can deliver similar performance as the para-virtualization based approach. Hence, in common cases, a guest OS integrated with vMitosis can experience the same performance benefits of gPT replication as if they were replicated with explicit hypervisor support.

Impact of misplaced gPT replicas (4KB pages): We acknowledge that our fully-virtualized approach may fail to achieve the best-case performance in some cases. While a vMitosis enabled guest OS can always discover the NUMA mappings of its vCPU's, the placement of gPT replica pages depends on the state of the hypervisor. Therefore, if the hypervisor fails to allocate gPT replicas from a vCPU's local socket (e.g., if free memory is not available),

Configuration	Data Allocation	ePT	gPT
OF	First-touch	First-touch	First-touch
OF+M(pv)	First-touch	Replicated	Replicated with para-virtualization
OF+M(fv)	First-touch	Replicated	Replicated with full virtualization

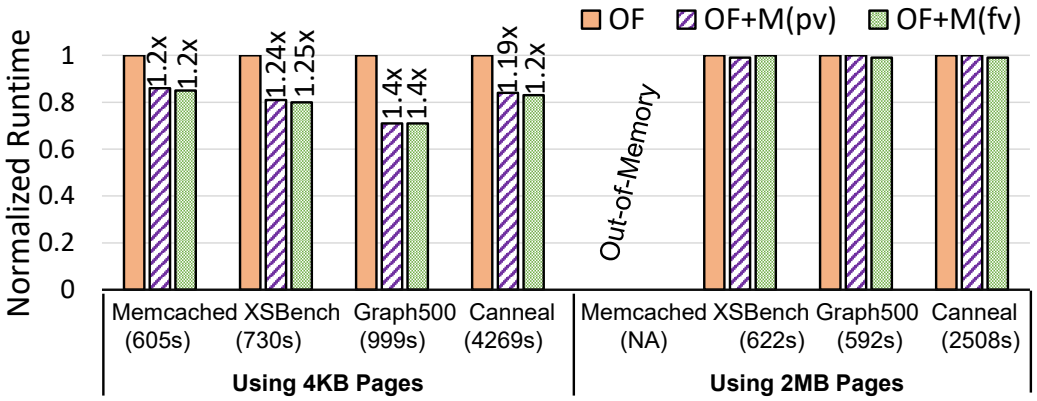


Figure 5.5: NUMA-oblivious: Workload performance, normalized to the base case (OF). Runtime for the base case in brackets. Numbers on top of the bars show speedup with vMitosis wherever significant. Configuration details are listed in the table at the top.

vMitosis may assign non-local gPT pages to some vCPUs. While we expect these cases to be rare, we still evaluate it to measure the worst-case overhead of non-local gPT replicas in our fully-virtualized approach OF+M(fv).

For these experiments, we artificially create a situation that mimics misplaced gPT replicas by configuring each thread's page table base register *cr*3 to point to one of the remote replicas. For example, we configure threads running on socket-0 to use socket-1's copy of the gPT and so on. This leads to 100% remote memory accesses for gPT. We then evaluate performance with and without replicating ePT.

Disabling ePT replication isolates the impact of misplaced gPT replicas. Our experiments showed a moderate 2%, 4%, and 5% slowdown over Linux/KVM for `Graph500`, `XSBench`, and `Memcached`, respectively. These results are in-line with our expectations: we do not expect high overheads since non-replicated page tables in Linux/KVM already result in about 75% remote gPT accesses on a 4-socket system, on average. If ePT replication is enabled, vMitosis outperforms Linux/KVM even if all gPT replicas are misplaced. This is also expected since vMitosis reduces the overall number of remote page walk accesses in this case – misplaced gPT adds 25% remote accesses but replicated ePT eliminates 75% remote accesses, on average.

Summary: We have shown that vMitosis improves application performance by replicating gPT and ePT in both NUMA-visible and NUMA-oblivious configurations. While 2MB pages perform well even without replicated page tables, they are susceptible to out-of-memory errors due to internal fragmentation.

5.4.3 Replication vs. migration of page tables

In Subsection 5.4.1, we profiled execution in a static setting by allocating data and page tables on different sockets and later migrating page tables close to data. However, scheduling policies of the OS/hypervisor have a dynamic effect on the placement of page tables. In this subsection, we demonstrate a live migration example using Memcached as a representative workload. We demonstrate live migration in both the guest OS and the hypervisor, and compare the effect of page table replication and migration in both these cases.

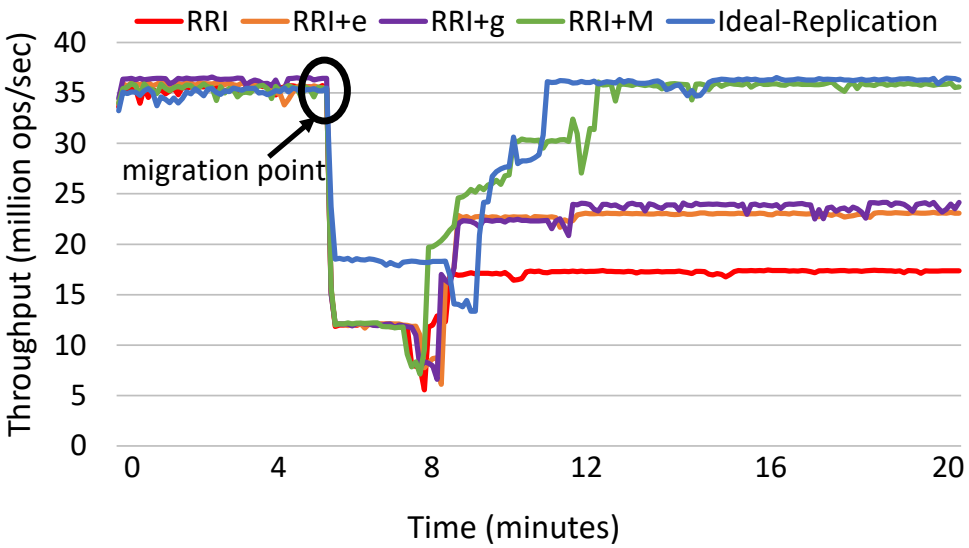
Evaluation methodology: For this evaluation, we select a Thin `Memcached` instance with a 30GB dataset and execute it in both NUMA-oblivious and NUMA-visible configurations. We initialize the cache and start querying the key-value store while measuring the throughput over time. After about five minutes, we migrate `Memcached` from Socket-0 to Socket-1. At this point, all memory accesses become remote for a few minutes until NUMA balancing starts migrating data pages.

Description of results: We first consider the NUMA-visible case (Figure 5.6a) and evaluate five configurations. Configurations RRI, RRI+e, RRI+g and RRIM+M are similar to the ones used in Subsection 5.4.1. Ideal-Replication represents pre-replicated page tables wherein page table accesses are always local.

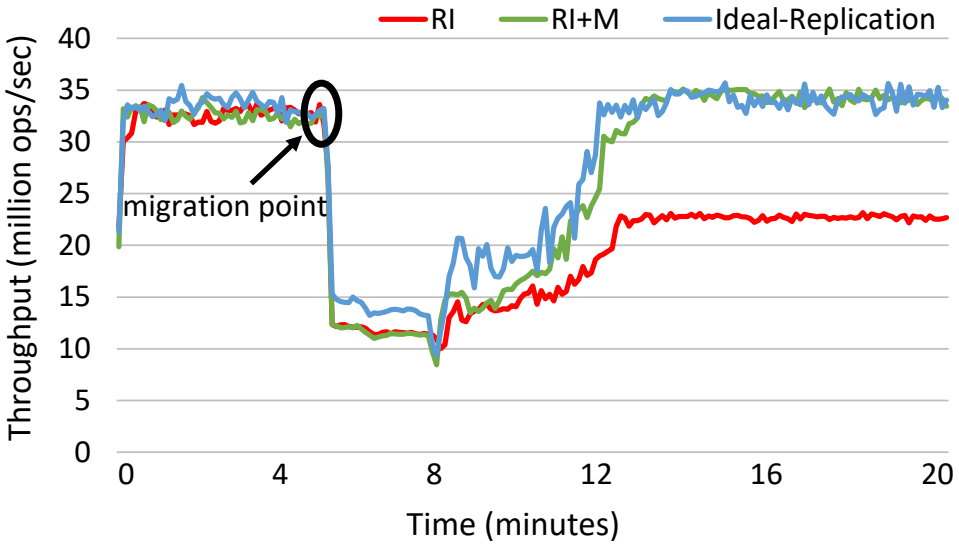
Initially, all five configurations operate at 35M operations per second, and then they experience a sharp drop in the throughput after `Memcached` gets migrated to another NUMA socket. In vanilla Linux/KVM (RRI) even after NUMA balancing has co-located the dataset of `Memcached`, the throughput is restored to only about 50% of its pre-migration level. When either ePT or gPT migration is enabled via vMitosis (RRI+e or RRI+g), we experience a similar pattern initially but reach 65% of the initial throughput in a few minutes. The best outcome is obtained with the migration of both ePT and gPT (RRI+M), where 100% of the throughput is regained. While the initial drop is less dramatic when using ideal pre-replicated page tables, our page table migration technique also quickly restores the throughput by migrating page tables with data pages. In the long run, vMitosis achieves the same behavior as ideal page table replication.

In the NUMA-oblivious case, the hypervisor-level NUMA balancing migrates guest data

Configuration	ePT
RI	Remote + Interference
RI+M	Migrated
Ideal-Replication	Replicated before migration



(a) **Workload migration by a NUMA-visible guest OS**



(b) **VM migration by KVM hypervisor**

Figure 5.6: Throughput of a Thin `Memcached` instance before, during and after migration. In the NUMA-visible case (a), the guest OS migrates `Memcached`. In the NUMA-oblivious case (b), the hypervisor migrates `Memcached`'s VM.

pages as well as the gPT when it migrates the VM. Hence, there are only three configurations in Figure 5.6b. Configuration RI represents the baseline Linux/KVM system where ePT is remote after VM migration. We evaluate vMitosis in two configuration where RI+M represents vMitosis with ePT migration and Ideal-Replication represents pre-replicated ePT.

Since gPT is automatically migrated by the hypervisor, the loss in Memcached's throughput in Linux/KVM in Figure 5.6b (RI) is lesser than the loss in the NUMA-visible case of Figure 5.6a (RRI): RI experiences $\approx$ 35% drop (local gPT, remote ePT) compared to 50% of RRI (remote gPT, remote ePT). However, this is still a significant performance loss considering that VM migration has completed from the system point of view. vMitosis restores the full performance by migrating ePT (RI+M); this behavior is also close to an ideal ePT replication scenario.

5.4.4 Memory and runtime overhead of vMitosis

We quantify the runtime overhead of our implementation with a micro-benchmark that invokes common memory management related system calls `mmap`, `mprotect`, and `munmap` – similar to the methodology discussed in Mitosis [39]. The micro-benchmark repeatedly invokes these system calls with different virtual memory region sizes. We measure throughput as the number of PTEs updated per second for each system call when invoked at 4KB, 4MB and 4GB granularity on three system configurations: Linux/KVM, vMitosis with migration, and vMitosis with replication. In vMitosis configurations, replication or migration of ePT and gPT is enabled simultaneously. Table 5.7 shows our measurements. We make the following observations based on these results.

First, the exact overhead of replication depends on the specifics of a particular system call. For example, the cost of `mmap` and `munmap` system calls is dominated by the time taken to allocate and free pages, respectively. In contrast, `mprotect` only updates certain page table bit, and therefore experiences significantly higher overhead due to replication. The overhead also depends on the size of the memory region. For smaller virtual memory area per system call, the overhead of context switching dominates that of updating page table replicas. Hence, the overheads are low in the 4KB case but more pronounced in the 4MB and 4GB cases.

Secondly, Linux/KVM and vMitosis (in its default migration mode) both maintain a single copy of the page tables. Hence, their throughput is roughly similar in all cases. Thin workloads, therefore, do not experience runtime overhead in vMitosis before or after migration or when they are never migrated across sockets. This is an important benefit that justifies the value of integrating page table migration with that of data pages. In contrast, replication-based page table migration (as in Mitosis) involves expensive page table updates. Furthermore, the

Syscall	Size	Linux/KVM	vMitosis (migration)	vMitosis (replication)
mmap	4KB	0.44	0.44 (1.0×)	0.40 (0.91×)
	4MB	1.10	1.10 (1.0×)	1.08 (0.98×)
	4GB	1.11	1.10 (1.0×)	1.08 (0.98×)
mprotect	4KB	0.82	0.83 (1.01×)	0.69 (0.84×)
	4MB	30.88	31.34 (1.01×)	9.05 (0.29×)
	4GB	31.82	31.81 (1.0×)	8.97 (0.28×)
munmap	4KB	0.34	0.34 (1.0×)	0.30 (0.88×)
	4MB	6.40	6.60 (1.03×)	4.92 (0.75×)
	4GB	6.62	6.64 (1.0×)	4.75 (0.72×)

Table 5.7: Throughput (measured as million PTEs updated per second) of different system calls when invoked with different virtual memory region sizes using 4KB mappings. Numbers in parentheses represent throughput normalized to Linux/KVM.

overhead of replication increases linearly with the number of replicas.

Third, through separate profiling of these overheads for ePT and gPT, we observe that the cost of updating gPT replicas dominates the overall replication overheads. In general, ePT updates are infrequent – ePT is updated when memory pages are first allocated to a VM which is a one time operation in the common case. Furthermore, the cost of updating ePT is dominated by VM-exits. Hence, ePT replication contributes only a marginal overhead in Table 5.7.

Finally, Table 5.8 shows the space overhead of vMitosis with different replication factors for a representative 1.5TiB workload using 4KB pages. For a typical densely populated address space, page tables consume a small fraction of overall memory (i.e., 0.2% – since a 4KB page table page maps 2MB of address space). Therefore, each 2D page table replica adds 0.4% memory overhead on virtualized systems, resulting in an overall 1.2% memory overhead on our four-socket system. With 2MB large pages, the space overhead of 4-way replication reduces to 36MB i.e., a negligible 0.003% of workload memory footprint. Overall, these space overheads are moderate compared to the performance benefits of vMitosis.

# replicas	ePT	gPT	Total
1	3GB	3GB	6GB (0.4%)
2	6GB	6GB	12GB (0.8%)
4	12GB	12GB	24GB (1.6%)

Table 5.8: Memory footprint of 2D page tables for a 1.5TB workload using 4KB pages with different replication factors. Numbers in parentheses represent memory consumption of page tables as a fraction of workload size.

5.4.5 Summary of results

We have shown that vMitosis fully mitigates the overheads of remote page table walks providing $1.8 - 3.1\times$ speedup for Thin workloads. It improves the performance of Wide workloads by $1.06-1.61\times$ in the NUMA-visible case and by $1.19-1.4\times$ in the NUMA-oblivious case. We have also shown that vMitosis can restore the performance of the `Memcached` server to its initial state within a few minutes of VM/workload migration. In a few cases, vMitosis provides significant improvement over Linux THP.

5.5 Discussion

5.5.1 Huge (large) pages

A major part of this dissertation is dedicated to showing that the overheads of address translation can be reduced to a great extent with efficient huge page management. Since huge pages minimize TLB misses, and avoid many DRAM accesses with shorter page tables, they can automatically minimize the effect of NUMA on address translation. However, integrating huge pages into existing systems has been extremely challenging [72, 75]. Some of the performance issues with huge pages appear due to inadequate OS-based huge page management algorithms as we have discussed in Chapter 2 and Chapter 3. This dissertation also takes several steps to remedy these issues. However, there are some issues that are fundamentally inherent in huge pages and no practical solutions are available to handle the associated challenges. Therefore, huge pages are no panacea and solutions like vMitosis are still desirable. We discuss three major challenges that make huge pages unsuitable for some applications or hard to use in practice.

First, huge pages negatively affect NUMA access locality for application data pages. With baseline 4KB pages, the OS can make fine-grained data placement/migration decisions based on application memory access patterns. This becomes difficult as the page size increases e.g., due to false sharing between threads executing on different NUMA sockets. These issues have been discussed in the academic literature [94] as well in the Linux kernel community [75]. A common solution to avoid NUMA locality issues is to fallback to smaller pages. This solution avoids NUMA penalty for data pages but brings page table access locality into play – the theme of this chapter. Therefore, vMitosis is useful when huge pages have a negative impact on application data page locality.

Second, huge pages risk increasing the memory footprint due to internal fragmentation in the virtual address space or unused baseline pages in an application. We discussed this issue in detail in Chapter 3. In this chapter also, we have shown that THP can cause out-of-memory (OOM) error for `Memcached` and `BTree` due to extra memory overhead. In HawkEye, we propose

a technique that can recover memory bloat from huge pages by de-duplicating unused baseline pages. Unfortunately, this solution also relies on breaking huge pages into regular 4KB pages.

Third, external physical memory fragmentation can prevent huge page allocations. In these cases, an OS falls back to using regular smaller pages. As discussed above and throughout this chapter, using small pages increases TLB misses and often necessitates high latency remote memory accesses for page table walks. Therefore, vMitosis is useful for long-running systems and when applications are susceptible to memory bloat with huge pages.

5.5.2 Shadow page tables

Under virtualization, address translation overheads can be reduced by replacing 2D page tables with shadow page tables. In shadow paging, hypervisor-managed shadow page tables translate guest virtual addresses directly to host physical addresses [182]. This reduces the maximium number of memory references involved in an address translation to only four (similar to native systems), as compared to 24 with two-level paging. However, shadow page tables must be kept consistent with guest page tables; for this, a typical hypervisor write protects gPT pages and applies gPT modifications to its shadow page tables. This process involves an expensive VM exit on every gPT update. Shadow page tables, therefore, involve a complicated trade-off i.e., optimizing hardware page table walks at the expense of high software memory management overheads.

Research has shown that TLB-intensive workloads that allocate memory once can benefit from shadow-paging [92]. vMitosis supports migration and replication of shadow page tables in KVM. Our experiences with shadow page tables have been mixed. In the best-case (when page table updates are infrequent), shadow paging combined with migration and replication with vMitosis improves performance by up to 2× over 2D page tables, at the expense of $2-6\times$ higher initialization time. In the worst-case, shadow paging degraded performance by more than 5×. We also observed extreme overheads due to guest kernel's services that update page tables (e.g., some of the workloads did not complete even in 24 hours when we enabled AutoNUMA in the guest OS). In general, shadow paging combined with the techniques of vMitosis could be useful for workloads that involve little kernel activities (e.g., HPC applications). The use of shadow paging in its current form does not seem fruitful, and consequently some hypervisors have abandoned it. However, techniques that exploit the best of shadow and extended paging have been explored in the literature [48, 92]; combined with vMitosis, such techniques could prove to be more powerful on heterogeneous memory systems.

5.6 Summary

In this chapter, we highlight that efficient placement of kernel objects is becoming important in NUMA systems. Our detailed analysis on a real platform shows that remote DRAM accesses due to misplaced guest and extended page-tables cause up to 3.1× slowdown. We present vMitosis – a system design to explicitly manage 2D page-tables in different virtualized environments. vMitosis leverages well-known replication and migration techniques and effectively eliminates NUMA effects on 2D page-table walks.

Chapter 6

Related work

Techniques to mitigate the overhead of virtual-to-physical address translation, and other memory management challenges discussed in this dissertation have been extensively studied in the literature. In this chapter, we discuss some important related works and highlight the key difference between prior work and our contributions.

Hardware techniques for efficient virtual memory: Hardware optimizations focus on reducing TLB misses and accelerating page table walks. Multiple page sizes supported by today's commercial CPUs provide both the benefits: fewer TLB misses and faster page table walks than regular 4KB pages [26]. Multi-level TLBs are used to balance the benefits of higher TLB coverage with the lookup latency of large caching structures [59]. Further, page walk caches are used to avoid accessing physical memory for higher levels of the page tables [51]. These hardware optimizations have found their way in commercial hardware available today. However, prior research and our work in this dissertation, have shown that despite these optimizations, large scale systems are still susceptible to the overhead of TLB misses.

Basu et. al. proposed direct segments to completely avoid TLB-miss processing cost through a special segmentation hardware [55]. In this scheme, processors use special registers to map some large contiguous virtual memory regions into physical memory using a BASE+OFFSET arithmetic. Direct segments require that each virtual address region using the segmentation technique is mapped contiguously in the physical address space. The same approach was later extended by Gandhi et. al. to virtualized environments [91], and by Haria et. al. to heterogeneous system architectures [104]. The primary focus of these solutions was to identify the necessary hardware support to use segmentation-based approach for address translation. All of these systems rely on contiguous physical memory – in many cases beyond the the size of conventional large pages. Therefore, these schemes are also susceptible to the effects of

fragmentation and solutions discussed in this dissertation are useful for such techniques.

For virtualized environments, shadow paging is another alternative to improve address translation performance [182]. With shadow paging, the hardware translates guest virtual address directly into host physical address, similar to native systems. This, however, requires the hypervisor to maintain a separate page table and keep it in sync with the guest page tables. For workloads that frequently update the page table, shadow paging incurs prohibitively high cost due to frequent VM exits. Gandhi et. al. proposed agile paging [92] to identify the best paging scheme at runtime e.g., agile paging uses nested page tables for address translation when page tables are updated frequently to avoid paying the cost of VM exits on each update. In other phases, it can use shadow paging to accelerate address translation.

OS-level challenges of memory fragmentation have also been considered in hardware designs: CoLT or Coalesced-Large-reach TLBs [149] were proposed to increase TLB reach using base pages, based on the observation that OSs naturally provide contiguous mappings at smaller granularity. CoLT uses the regular page size but encodes contiguity hints in the spare bits of page tables. This way, it effectively achieves higher TLB coverage but without increasing the page size. This approach was further extended to page walk caches and huge pages [60, 76, 150]. In fact, techniques to support huge pages in non-contiguous physical memory have also been proposed [88].

POM-TLB services a TLB miss using a single memory lookup with a large in-memory TLB, and further leverages regular data caches to speed up address translation [159]. SpecTLB speculatively provides address translation on a TLB miss by guessing virtual-to-physical address mappings [52]. ASAP prefetches translations to reduce page walk latency to that of a single memory lookup [132]. It first orders page table pages to match that of the virtual memory pages and then uses a BASE+OFFSET arithmetic that directly indexes into the page tables. Tailored page sizes use whatever contiguity OS can afford to allocate [103]. Park et. al. proposed to use large pages to allocate page table entries to reduce the height of the page table [148]. Various application specific and machine learning techniques have also been explored for accelerating address translation [42, 131].

One major limitation of current address translation schemes is that they inherently make page table walks a pointer chasing operation through multiple layers of the radix-based page table tree. Therefore, multiple memory requests involved in a page table walk cannot benefit from hardware support for memory-level parallelism. Hashing-based page tables avoid this bottleneck of radix tree and have been proposed as an alternate address translation structure [187]. Skarlatos et. al. recently demonstrated the benefits of using elastic cuckoo hash table as the page table data structure [165]. Elastic cuckoo page tables convert pointer-chasing operations

into fully parallel lookups, support efficient page table resizing and memory sharing across different processes.

Other hardware optimizations include: redundant memory mappings (RMMs) proposed by Gandhi et. al. [115], and OS/architecture co-design approach proposed by Alverti et. al. [44]. These solutions have many conceptual similarities. In the former, the hardware uses an additional translation structure called range TLB for virtual-to-physical address translation, albeit at an arbitrary granularity – not using fixed size pages. Range TLB is much smaller in size than regular TLBs and are populated by walking the in-memory range tables. Range TLB is looked up in parallel with regular TLBs by the hardware. In the latter approach, a similar contiguity-aware paging scheme is employed to speed up address translation in virtualized environments.

Software techniques for efficient virtual memory: Early work discussed multiple ways of supporting huge pages in HP-UX OS [171]. Navarro et al. implemented support for multiple page sizes with reservation-based huge page allocations and contiguity-aware page reclamation [139]. FreeBSD's huge page management is largely influenced by this seminal work. In Chapter 3, we discussed the pitfalls associated with the reservation-based approach in detail. Similar issues have been highlighted in a recent work [121].

Ingens is perhaps the most closely related work to the contributions of this dissertation. Ingens proposes to mix the aggressive large page allocation scheme of Linux with the conservative approach of FreeBSD. This way, Ingens improves THP support in terms of fairness, bloat and latency by tracking huge page utilization and access frequency of pages. We quantitatively compare our fine-grained huge page management solution HawkEye with Ingens in Chapter 3. Recent proposal Quicksilver uses hybrid strategies across different stages in the lifetime of a large page [190]. Specifically, it employs aggressive allocation, hybrid preparation, relaxed mapping creation, on-demand mapping destruction and preemptive deallocation to improve performance and lower latency and memory bloat.

As discussed in Chapter 3 of this dissertation, some part of the address space may be subject to relatively higher TLB miss overheads in some applications. To maximize the benefits of huge pages, compiler or application hints can also be used to assist the OS in prioritizing huge page mappings for such regions [41, 129]. Linux already provides an interface to communicate such preferences through the `madvise` system call [6]. In this dissertation, we also show that user-space hints are not necessarily required. Using the page table access bits, the OS can estimate the profitability of huge pages and allocate memory contiguity accordingly. This way, our approach is inline with the concept of transparent huge pages.

Carrefour-LP [94] highlighted that large pages can increases remote memory accesses in

NUMA systems. The authors highlight the effect of page size on load balancing across multiple memory controllers, one per NUMA node. The authors use hardware performance counters to detect when large pages affect data access locality and load balancing, and propose disabling large pages if they are predicted to harm performance. Guo et al. proposed proactively breaking huge pages to improve memory efficiency via page sharing in virtualized environments [101]. Ingens and SmartMD [121, 100] provide alternative fine-grained mechanisms to moderate the effect of large pages on page sharing while still trying to achieve high address translation performance.

Translation Ranger proposed a new OS service that actively creates contiguity in physical memory [186]. This functionality, however, requires application modifications and is unsuitable for applications that incrementally allocate memory. Some part of this work provided a preliminary patch set to extend Linux's THP support to 1GB pages [184]. Their approach, however, always prefer promoting address regions to 1GB pages through a background service – even if contiguous memory is available during page faults. Consequently, it incurs unnecessary data movement even when physical memory is un-fragmented or moderately fragmented. In large-memory systems, this could amount to migrating terabytes of data to allocate 1GB pages. Further, the inability to immediately deploy 1GB pages, even when it was possible, leaves significant performance on the table. In contrast, our proposal Trident employs 1GB pages aggressively without requiring application modifications as discussed in Chapter 4.

Techniques to deal with memory fragmentation: Memory allocation and reclamation decisions guided by the size or expected lifetime of objects form the basis for many copying or generational garbage collectors in user space [125, 79, 170, 83, 50, 169]. Recent work on C++ memory allocator uses a pure lifetime prediction algorithm to minimize internal fragmentation in application virtual address space, especially to alleviate memory overhead while using large pages [128]. Unfortunately, such solutions are not well-suited for mitigating kernel-level fragmentation because the kernel has to deal with unmovable pages. Moreover, application specific knowledge (e.g., object lifetime) is, in general, not available to OSs [160, 96].

S. Kim et al. [116] proposed proactive anti-fragmentation that allocates contiguous memory to each process from the beginning. This way, contiguous memory can be recovered when a process exits or gets killed by the out-of-memory killer. Their solution, however, is specific to Android due to its relatively short-lived processes. Moreover, processes are killed in Android under memory pressure as compared to workstations or servers where paging is preferred over killing a process.

Gorman et al. proposed defragmenting memory with Lumpy reclaim [98]. Contiguity-aware page replacement discussed by Navarro et al. [139] is also similar to Lumpy reclaim. These

techniques try to recover memory contiguity by dropping file-backed pages from contiguous regions. Lumpy reclaim was merged in Linux but was removed later as it created regressions due to additional I/O traffic [5]. In current Linux versions, memory compaction is the preferred mechanism for defragmenting memory [69]. We show that compaction leads to severe problems in its current form. Illuminator solves compaction related problems by carefully managing unmovable pages.

Gorman et al. also proposed two different anti-fragmentation schemes to aid compaction [97], both of which are merged into Linux. One of the approaches i.e., zone-based demands a static partitioning of memory between movable and unmovable regions at boot time. However, such an approach is difficult to employ when memory capacity is limited or when the workload characteristics are not known in advance. The other approach i.e., Grouping Pages Based on their Mobility type (GPBM [96]) manages the size of memory partitions dynamically and is more suitable for dynamic workloads. Page-clustering algorithm of Linux discussed in Chapter 2 is largely influenced by GPBM. We show how page-clustering leads to fragmentation via pollution in its current form, and effectively handle the associate challenges with Illuminator.

Our previous work [143] inspired the design and implementation of Illuminator. Illuminator addresses several shortcomings of this prior work: 1) page allocation in Illuminator is a constant time O(1) operation, comparing to an $O(n)$ algorithm of [143] (where n is the number of pageblocks). O(n) algorithm is prohibitively expensive for performance critical subsystems such as the kernel page allocator. (2) this previous work relied on the two-way classification of pageblocks whose shortcomings have been discussed throughout Chapter 2. Illuminator shows that cross subsystem visibility of unmovable pages and coordination among several layers (e.g., buddy allocator, slab allocator, and compaction) is crucial. Moreover, Illuminator has also been evaluated across native and virtualized systems with respect to various aspects of performance such as latency, OS jitter and execution speedup.

NUMA optimizations: Software-based NUMA optimizations have been proposed for decades [179, 63]. Applications with complex NUMA properties can directly leverage system calls to retain control over the placement of their threads and data [123]. Other applications can rely on built-in system heuristics that automatically drive NUMA optimizations [90, 141, 73]. While OS and library support thus far has focused only on user data pages, we have shown that the placement of important kernel structures such as page tables is also crucial for performance. Recent proposals such as KLOC [112] and nrOS [57] also explore the effect of non-uniform memory access lantecy on the performance of different kernel objects and subsystems.

Replication and migration are popular techniques that have been applied in various contexts [167]. Carrefour is a kernel design that transparently replicates and migrates application

data on NUMA systems [78]. Shoal provides an abstraction to replicate, partition, and distribute arrays across NUMA domains [109]. Carrefour-LP optimized large page performance on NUMA systems by judiciously allocating/demoting large pages [94]. RadixVM used replication to improve the scalability of the xv6 kernel [68]. Various locking techniques also employ replication for reducing cross-CPU traffic to achieve high scalability [66]. In contrast, we apply replication and migration to improve address translation performance. Our technique of discovering the server's NUMA topology in vMitosis is inspired by Smelt that derived efficient communication patterns on multi-core platforms via online measurements [110].

In contrast to conventional NUMA optimizations, vMitosis explores the effect of memory heterogeneity on virtualized page table walks. While prior techniques are effective for application data pages, vMitosis can be used to further improve their efficiency on NUMA-like systems.

Chapter 7

Conclusion and Looking Forward

Data-centric computation in the era of AI stresses the memory subsystem of computers wherein the efficiency of computation is often determined by that of data accesses. However, before data can be accessed, the processor needs to translate application-generated virtual addresses into physical addresses where the data actually resides. This act of address translation often consumes a significant fraction of CPU cycles. In this thesis, we proposed four optimizations in the operating system and hypervisor to moderate the overheads of address translation in a manner that requires no change in applications or the hardware.

In the major part of the thesis, we explored various challenges involved in the management of huge pages. We highlight various shortcomings of current system designs in managing huge pages. Our proposal Illuminator deals with external physical memory fragmentation effectively in turn making it feasible for an OS to allocate huge pages. In HawkEye, we proposed various techniques to maximize the performance benefits of huge pages and handle various memory management trade-offs. Trident extends OS support for transparent huge pages to harness all page sizes available on modern x86 systems. Finally, in vMitosis, we propose NUMA-aware page table management to mitigate the effect of non-uniform memory access latency on address translation.

Overall, this thesis shows that virtual memory can continue to thrive even in the age of data-centric computation. Moreover, we show that this can be done with robust memory management in operating systems and hypervisors while avoiding modifications to user applications and additional hardware features. We open-sourced all software created as a part of this thesis [27, 28, 29, 30].

Looking ahead, we foresee that memory management will gain further importance for data-centric computation with the advent of new memory technologies such as non-volatile memory, high-bandwidth die-stacked memory, and network-attached dis-aggregated memory. An im-

portant research direction is to explore how virtual memory should abstract the heterogeneity across memory technologies. Further, heterogeneity is not just limited to the memory subsystem. Compute accelerators such as GPUs are playing a key role in processing huge datasets. These trends pose new research challenges in managing the heterogeneity in both memory and compute.

Bibliography

[1] Arch Linux becomes unresponsive from khugepaged. http://unix.stackexchange.com/questions/161858/arch-linux-becomes-unresponsive-from-khugepaged. 38

[2] Recommendation for disabling huge pages for Hadoop. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Hadoop_Tuning_Guide-Version5.pdf. 6, 9, 38

[3] The black magic of systematically reducing linux os jitter. http://highscalability.com/blog/2015/4/8/the-black-magic-of-systematically-reducing-linux-os-jitter.html. 38

[4] khugepaged eating 100% cpu. https://bugzilla.redhat.com/show_bug.cgi?id=879801. 38

[5] Removal of lumpy reclaim. https://lwn.net/Articles/488993/. 132

[6] Transparent Hugepage Support. https://www.kernel.org/doc/Documentation/vm/transhuge.txt. 42, 130

[7] Recommendation for disabling huge pages for MongoDB. https://docs.mongodb.org/manual/tutorial/transparent-huge-pages/. 6, 9, 38

[8] Recommendation for disabling huge pages for NuoDB. http://www.nuodb.com/techblog/linux-transparent-huge-pages-jemalloc-and-nuodb. 6, 38

[9] About the Virtual Memory System. https://developer.apple.com/library/content/documentation/Performance/Conceptual/ManagingMemory/Articles/AboutMemory.html. 36

[10] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. 74

[11] pgbench. https://www.postgresql.org/docs/9.1/static/pgbench.html. 27

[12] Recommendation for disabling huge pages for Redis. http://redis.io/topics/latency. 6, 9, 38, 53, 101

[13] Mmtests: Benchmarking framework primarily aimed at linux kernel testing. https://github.com/gormanm/mmtests. 29

[14] sysbench. https://dev.mysql.com/downloads/benchmarks.html. 27

[15] Tales from the Field: Taming Transparent Huge Pages on Linux. https://www.perforce.com/blog/151016/tales-field-taming-transparent-huge-pages-linux. 38

[16] Why TokuDB Hates Transparent HugePages. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Hadoop_Tuning_Guide-Version5.pdf. 38

[17] Recommendation for disabling huge pages for VoltDB. https://docs.voltdb.com/AdminGuide/adminmemmgt.php. 6, 9, 38

[18] Clearing pages in the idle loop. https://www.mail-archive.com/freebsd-hackers@freebsd.org/msg13993.html, 2000. 47

[19] Linux: Page zeroing strategy. https://yarchive.net/comp/linux/page_zeroing_strategy.html, 2000. 47

[20] GUPS: HPCC RandomAccess benchmark, 2006. https://github.com/alexandermerritt/gups. 74, 101

[21] Memory part 5: What programmers can do. https://lwn.net/Articles/255364/, 2007. 48

[22] Mysteries of windows memory management revealed: Part two. https://blogs.msdn.microsoft.com/tims/2010/10/29/pdc10-mysteries-of-windows-memory-management-revealed-part-two/, 2010. 47

[23] Coral benchmark codes. https://asc.llnl.gov/CORAL-benchmarks/#hacc, 2014. 67

[24] Remove pg_zero and zeroidle (page-zeroing) entirely. https://news.ycombinator.com/item?id=12227874, 2016. 47

[25] C++ associative containers. https://github.com/sparsehash/sparsehash, 2016. 67

[26] Hugepages. https://wiki.debian.org/Hugepages, 2017. 38, 48, 128

[27] Illuminator. https://github.com/apanwariisc/Illuminator, 2018. 8, 134

[28] Hawkeye. https://github.com/apanwariisc/HawkEye, 2019. 8, 134

[29] Trident. https://github.com/csl-iisc/Trident-MICRO21-artifact, 2019. 8, 134

[30] vmitosis. https://github.com/mitosis-project/vmitosis-asplos21-artifact, 2019. 8, 134

[31] Cgroups: Linux programmer's manual. http://man7.org/linux/man-pages/man7/cgroups.7.html, 2019. 53

[32] Freebsd: Pre-faulting and zeroing optimizations. https://www.freebsd.org/doc/en_US.ISO8859-1/articles/vm-design/prefault-optimizations.html, 2019. 47

[33] Transparent hugepage support. https://www.kernel.org/doc/Documentation/vm/transhuge.txt, 2019. 42

[34] ab - apache http server benchmarking tool, 2021. https://httpd.apache.org/docs/2.4/programs/ab.html. 80

[35] Intel® microarchitecture code named skylake events, 2021. https://download.01.org/perfmon/index/skylake.html. 74

[36] Libsvm data: Classification (binary class), 2021. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html. 74

[37] Wikichip: Intel sandy bridge microarchitecture, 2021. https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake. 72

[38] Wikichip: Intel sandy bridge microarchitecture, 2021. https://en.wikichip.org/wiki/intel/microarchitectures/sandy_bridge_(client). 71

[39] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently self-replicating page-tables for large-memory machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 283–300, Lausanne,

Switzerland, 2020. ISBN 9781450371025. doi: 10.1145/3373376.3378468. URL https://doi.org/10.1145/3373376.3378468. 5, 99, 106, 123

[40] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 631–644, Xi'an, China, 2017. ISBN 9781450344654. doi: 10.1145/3037697.3037706. URL https://doi.org/10.1145/3037697.3037706. 98

[41] Mohammad Agbarya, Idan Yaniv, and Dan Tsafrir. Memomania: From huge to huge-huge pages. In *Proceedings of the 11th ACM International Systems and Storage Conference*, SYSTOR '18, pages 112–112, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5849-1. doi: 10.1145/3211890.3211918. URL http://doi.acm.org/10.1145/3211890.3211918. 48, 130

[42] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. Do-it-yourself virtual memory translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 457–468, Toronto, ON, Canada, 2017. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080209. URL http://doi.acm.org/10.1145/3079856.3080209. 98, 129

[43] K. Albayraktaroglu, A. Jaleel, Xue Wu, M. Franklin, B. Jacob, Chau-Wen Tseng, and D. Yeung. Biobench: A benchmark suite of bioinformatics applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, ISPASS '05, pages 2–9, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7803-8965-4. doi: 10.1109/ISPASS.2005.1430554. URL http://dx.doi.org/10.1109/ISPASS.2005.1430554. 27, 53

[44] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Enhancing and exploiting contiguity for fast memory virtualization. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 515–528, 2020. doi: 10.1109/ISCA45697.2020.00050. 130

[45] Nadav Amit, Dan Tsafrir, and Assaf Schuster. Vswapper: A memory swapper for virtualized environments. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14,

pages 349–366, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541969. URL http://doi.acm.org/10.1145/2541940.2541969. 68

[46] AnandTech. The ice lake benchmark preview: Inside intel's 10nm, 2019. https://www.anandtech.com/show/14664/testing-intel-ice-lake-10nm/2. 72

[47] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H. Loh. Avoiding tlb shootdowns through self-invalidating tlb entries. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 273–287, 2017. doi: 10.1109/PACT.2017.38. 29

[48] Chang S. Bae, John R. Lange, and Peter A. Dinda. Enhancing virtualized application performance through dynamic adaptive paging mode selection. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, page 255–264, Karlsruhe, Germany, 2011. ISBN 9781450306072. doi: 10.1145/1998582.1998639. URL https://doi.org/10.1145/1998582.1998639. 126

[49] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks;summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM. ISBN 0-89791-459-7. doi: 10.1145/125826.125925. URL http://doi.acm.org/10.1145/125826.125925. 27, 45, 53, 69, 74

[50] Katherine Barabash, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, and Erez Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Trans. Program. Lang. Syst.*, 27(6):1097–1146, November 2005. ISSN 0164-0925. doi: 10.1145/1108970.1108972. URL http://doi.acm.org/10.1145/1108970.1108972. 131

[51] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 48–59, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. doi: 10.1145/1815961.1815970. URL http://doi.acm.org/10.1145/1815961.1815970. 128

[52] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Spectlb: A mechanism for speculative address translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 307–318, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0472-6. doi: 10.1145/2000064.2000101. URL http://doi.acm.org/10.1145/2000064.2000101. 129

[53] A. Basu, M. D. Hill, and M. M. Swift. Reducing memory reference energy with opportunistic virtual caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 297–308, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4503-1642-2. URL http://dl.acm.org/citation.cfm?id=2337159.2337194. 3

[54] Arkaprava Basu. *Revisiting virtual memory*. PhD thesis, University of Wisconsin-Madison, December 2013. 80

[55] 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 '13, pages 237–248, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. doi: 10.1145/2485922.2485943. URL http://doi.acm.org/10.1145/2485922.2485943. 3, 10, 12, 37, 53, 128

[56] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015. URL http://arxiv.org/abs/1508.03619. 53, 69, 74

[57] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. Nros: Effective replication and sharing in an operating system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 295–312. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/bhardwaj. 132

[58] Ravi Bhargava, Ben 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, Seattle, WA, USA, March 1-5, 2008*, pages 26–35, 2008. doi: 10.1145/1346281.1346286. URL http://doi.acm.org/10.1145/1346281.1346286. 3, 4, 38

[59] A. Bhattacharjee, D. Lustig, and M. Martonosi. *Architectural and Operating System Support for Virtual Memory*. Synthesis Lectures on Computer Architecture. Morgan &

Claypool Publishers, 2017. ISBN 9781627059336. URL https://books.google.co.in/books?id=roM4DwAAQBAJ. 4, 128

[60] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 383–394, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2638-4. doi: 10.1145/2540708.2540741. URL http://doi.acm.org/10.1145/2540708.2540741. 38, 129

[61] Abhishek Bhattacharjee. Translation-Triggered Prefetching. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 63–76, Xi'an, China, 2017. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037705. URL http://doi.acm.org/10.1145/3037697.3037705. 98, 100

[62] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011. 27, 53, 69, 74, 101

[63] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. Simple but Effective Techniques for NUMA Memory Management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 19–31, Litchfield Park, AZ, USA, 1989. ISBN 0-89791-338-8. doi: 10.1145/74850.74854. URL http://doi.acm.org/10.1145/74850.74854. 5, 132

[64] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005. ISBN 0596005652. 11, 49

[65] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 157–166, Shenzhen, China, 2013. ISBN 9781450319225. doi: 10.1145/2442516.2442532. URL https://doi.org/10.1145/2442516.2442532. 111

[66] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 207–221, Xi'an, China, 2017. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037721. URL http://doi.acm.org/10.1145/3037697.3037721. 98, 133

[67] Josiah L. Carlson. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA, 2013. ISBN 1617290858, 9781617290855. 41, 74

[68] 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 '13, pages 211–224, Prague, Czech Republic, 2013. ISBN 978-1-4503-1994-2. doi: 10.1145/2465351.2465373. URL http://doi.acm.org/10.1145/2465351.2465373. 133

[69] Jonathan Corbet. Memory compaction. https://lwn.net/Articles/368869/, . 10, 14, 39, 132

[70] Jonathan Corbet. Proactive compaction. https://lwn.net/Articles/717656/, . 10

[71] Jonathan Corbet. Virtually mapped kernel stacks. https://lwn.net/Articles/692208/, . 12

[72] Jonathan Corbet. Large pages, large blocks, and large problems. Online https://lwn.net/Articles/250335/, 2007. 6, 125

[73] Jonathan Corbet. AutoNUMA: the other approach to NUMA scheduling. https://lwn.net/Articles/488709/, 2012. 108, 132

[74] Jonathan Corbet. Numa scheduling progress. Online https://lwn.net/Articles/568870/, 2014. 98, 101

[75] Jonathan Corbet. Transparent huge pages, numa locality, and performance regressions. Online https://lwn.net/Articles/787434/, 2019. 6, 125

[76] Guilherme Cox and Abhishek Bhattacharjee. Efficient address translation for architectures with multiple page sizes. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 435–448, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037704. URL http://doi.acm.org/10.1145/3037697.3037704. 38, 129

[77] Ian Cutress. Intel's Enterprise Extravaganza 2019: Launching Cascade Lake, Optane DCPMM, Agilex FPGAs, 100G Ethernet, and Xeon D-1600. https://www.anandtech.com/show/14155/intels-enterprise-extravaganza-2019-roundup, 2019. 72

[78] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 381–394, Houston, Texas, USA, 2013. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451157. URL http://doi.acm.org/10.1145/2451116.2451157. 5, 98, 133

[79] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 261–269, New York, NY, USA, 1990. ACM. ISBN 0-89791-343-4. doi: 10.1145/96709.96735. URL http://doi.acm.org/10.1145/96709.96735. 131

[80] Linux Kernel Documentation. Split page table lock. Online https://www.kernel.org/doc/html/latest/vm/split_page_table_lock.html, 2020. 109

[81] Windows documentation. Large-Page Support in Windows. https://msdn.microsoft.com/en-us/library/windows/desktop/aa366720(v=vs.85).aspx. 20

[82] Windows Documentation. Large-Page Support in Windows Operating System. https://docs.microsoft.com/en-us/windows/win32/memory/large-page-support. 10

[83] Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 274–284, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349336. URL http://doi.acm.org/10.1145/349299.349336. 131

[84] E Knuth Donald et al. The art of computer programming. *Sorting and searching*, 3: 426–458, 1999. 81

[85] 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, pages 229–237, Berkeley, CA, USA, 1999. USENIX Association. ISBN 1-880446-39-1. URL http://dl.acm.org/citation.cfm?id=296806.296833. 47

[86] Niall Douglas. User mode memory page allocation: A silver bullet for memory allocation? *CoRR*, abs/1105.1811, 2011. URL http://arxiv.org/abs/1105.1811. 47

[87] Ulrich Drepper. What every programmer should know about memory, 2007. 47

[88] Yu Du, Miao Zhou, Bruce R. Childers, Daniel Mossé, and Rami Melhem. Supporting superpages in non-contiguous physical memory. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 223–234, 2015. doi: 10.1109/HPCA.2015.7056035. 129

[89] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5, August 2004. ISSN 1075-3583. 74, 101

[90] FreeBSD. FreeBSD - NUMA. https://wiki.freebsd.org/NUMA. 132

[91] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 178–189, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-6998-2. doi: 10.1109/MICRO.2014.37. URL http://dx.doi.org/10.1109/MICRO.2014.37. 3, 5, 10, 37, 38, 128

[92] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 707–718, Seoul, Republic of Korea, 2016. ISBN 978-1-4673-8947-1. doi: 10.1109/ISCA.2016.67. URL https://doi.org/10.1109/ISCA.2016.67. 5, 98, 126, 129

[93] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on numa systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 231–242, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-10-2. URL http://dl.acm.org/citation.cfm?id=2643634.2643659. 38

[94] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on numa systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 231–242, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-10-2. URL http://dl.acm.org/citation.cfm?id=2643634.2643659. 125, 130, 133

[95] Mel Gorman. Cost model for memory compaction. http://lkml.iu.edu/hypermail/linux/kernel/1211.2/03725.html. 28

[96] Mel Gorman and Patrick Healy. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 41–50, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-134-7. doi: 10.1145/1375634.1375641. URL http://doi.acm.org/10.1145/1375634.1375641. 9, 14, 36, 98, 131, 132

[97] Mel Gorman and Andy Whitcroft. The what, the why and the where to of anti-fragmentation. In *Linux Symposium*, page 141, 2006. 14, 75, 132

[98] Mel Gorman and Andy Whitcroft. Supporting the allocation of large contiguous regions of memory. In *Linux Symposium*, page 141, 2007. 131

[99] Brendan Gregg. How netflix tunes ec2 instances for performance, 2017. http://www.brendangregg.com/Slides/AWSreInvent2017_performance_tuning_EC2.pdf. 94

[100] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John C. S. Lui. Smartmd: A high performance deduplication engine with mixed pages. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 733–744, Berkeley, CA, USA, 2017. USENIX Association. ISBN 978-1-931971-38-6. URL http://dl.acm.org/citation.cfm?id=3154690.3154759. 50, 131

[101] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. Proactively breaking large pages to improve memory overcommitment performance in vmware esxi. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 39–51, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3450-1. doi: 10.1145/2731186.2731187. URL http://doi.acm.org/10.1145/2731186.2731187. 131

[102] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. Proactively breaking large pages to improve memory overcommitment performance in vmware esxi. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 39–51, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3450-1. doi: 10.1145/2731186.2731187. URL http://doi.acm.org/10.1145/2731186.2731187. 50, 53

[103] Faruk Guvenilir and Yale N Patt. Tailored page sizes. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 900–912. IEEE, 2020. 129

[104] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Devirtualizing memory in heterogeneous systems. *SIGPLAN Not.*, 53(2):637–650, mar 2018. ISSN 0362-1340. doi: 10.1145/3296957.3173194. URL https://doi.org/10.1145/3296957.3173194. 128

[105] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006. ISSN 0163-5964. doi: 10.1145/1186736.1186737. URL http://doi.acm.org/10.1145/1186736.1186737. 27, 53, 69

[106] 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 *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 257–273. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/hunter. 4, 7, 72

[107] Intel. Intel® 64 and ia-32 architectures developer's manual, vol. 3c. Online https://www.intel.in/content/www/in/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.html, 2020. 110

[108] Intel Corporation. 5-Level Paging and 5-Level EPT. https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf, 2017. 5

[109] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. Shoal: Smart Allocation and Replication of Memory for Parallel Programs. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 263–276, Santa Clara, CA, 2015. ISBN 978-1-931971-225. URL https://www.usenix.org/conference/atc15/technical-session/presentation/kaestle. 133

[110] Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, and Timothy Roscoe. Machine-aware atomic broadcast trees for multicores. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 33–48, Savannah, GA, USA, 2016. ISBN 9781931971331. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kaestle. 111, 133

[111] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 521–534, Toronto, ON, Canada, 2017. ISBN 9781450348928. doi: 10.1145/3079856.3080245. URL https://doi.org/10.1145/3079856.3080245. 5, 98

[112] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. *KLOCs: Kernel-Level Object Contexts for Heterogeneous Memory Systems*, page 65–78. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450383172. URL https://doi.org/10.1145/3445814.3446745. 132

[113] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal. Energy-efficient address translation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 631–643, March 2016. doi: 10.1109/HPCA.2016.7446100. 3

[114] Vasileios Karakostas, Osman S. Unsal, Mario Nemirovsky, Adrián Cristal, and Michael M. Swift. Performance analysis of the memory management unit under scale-out workloads. *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12, 2014. xv, 46, 49

[115] 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. *SIGARCH Comput. Archit. News*, 43(3S):66–78, jun 2015. ISSN 0163-5964. doi: 10.1145/2872887.2749471. URL https://doi.org/10.1145/2872887.2749471. 130

[116] Sang-Hoon Kim, Sejun Kwon, Jin-Soo Kim, and Jinkyu Jeong. Controlling physical memory fragmentation in mobile systems. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 1–14, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3589-8. doi: 10.1145/2754169.2754179. URL http://doi.acm.org/10.1145/2754169.2754179. 10, 131

[117] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, oct 1965. ISSN 0001-0782. doi: 10.1145/365628.365655. URL https://doi.org/10.1145/365628.365655. 15

[118] Donald E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., third edition, 1997. ISBN 0201896834 9780201896831. 15

[119] Joe Kozlowicz. Checking your virtual machine numa configuration. Online https://www.greenhousedata.com/blog/dont-turn-numa-numa-yay-into-numa-numa-nay-checking-virtual-machine-numa, 2018. 99

[120] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselý, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. Latr: Lazy translation coherence. *SIGPLAN Not.*, 53(2):651–664, mar 2018. ISSN 0362-1340. doi: 10.1145/3296957.3173198. URL https://doi.org/10.1145/3296957.3173198. 17, 29

[121] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 705–721, GA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon. 3, 38, 39, 40, 52, 53, 66, 91, 98, 130, 131

[122] Kieran Laffan. Sql server best practices, part i: Configuration. Online https://www.varonis.com/blog/sql-server-best-practices-part-configuration/, 2020. 99

[123] Christoph Lameter. NUMA (Non-Uniform Memory Access): An Overview. *Queue*, 11 (7):40:40–40:51, July 2013. ISSN 1542-7730. doi: 10.1145/2508834.2513149. URL http://doi.acm.org/10.1145/2508834.2513149. 5, 132

[124] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page fault support for network controllers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 449–466, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344654. doi: 10.1145/3037697.3037710. URL https://doi.org/10.1145/3037697.3037710. 36

[125] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983. ISSN 0001-0782. doi: 10.1145/358141.358147. URL http://doi.acm.org/10.1145/358141.358147. 131

[126] Linux Kernel Documentation. 5-Level Paging. https://www.kernel.org/doc/html/latest/x86/x86_64/5level-paging.html, 2022. 5

[127] LinuxVersions. Linux kernel changelog. https://kernelnewbies.org/LinuxVersions. 10

[128] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. *Learning-Based Memory Allocation for C++ Server Workloads*, page 541–556. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450371025. URL https://doi.org/10.1145/3373376.3378525. 40, 42, 131

[129] Joshua Magee and Apan Qasem. A case for compiler-driven superpage allocation. In *Proceedings of the 47th Annual Southeast Regional Conference*, ACM-SE 47, pages 82:1–82:4, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-421-8. doi: 10.1145/1566445.1566553. URL http://doi.acm.org/10.1145/1566445.1566553. 45, 130

[130] Kristie Mann. Five use cases of intel optane dc persistent memory at work in the data center, 2019. https://itpeernetwork.intel.com/intel-optane-use-cases/. 72

[131] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Virtual address translation via learned page table indexes. In *Proceedings of the Workshop on ML for Systems at NeurIPS*, 2018. 129

[132] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched address translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 1023–1036, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358294. URL https://doi.org/10.1145/3352460.3358294. 129

[133] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. https://www.cs.virginia.edu/stream/, 2019. 103

[134] Paul E McKenney, Dipankar Sarma, Ingo Molnar, and Suparna Bhattacharya. Extending rcu for realtime and embedded workloads. In *Ottawa Linux Symposium, pages v2*, pages 123–138. Citeseer, 2006. 18

[135] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2nd edition, 2014. ISBN 0321968972, 9780321968975. 36

[136] Timothy Merrifield and H. Reza Taheri. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, pages 25–35, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3947-6. doi: 10.1145/2892242.2892258. URL http://doi.acm.org/10.1145/2892242.2892258. 3, 5

[137] Jeffrey C. Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating system support for nvm+dram hybrid main memory. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, page 14, Monte Verità, Switzerland, 2009. 5, 97

[138] Djob Mvondo, Boris Teabe, Alain Tchana, Daniel Hagimont, and Noel De Palma. Memory flipping: a threat to NUMA virtual machines in the cloud. In *2019 IEEE Conference on Computer Communications, INFOCOM 2019, Paris, France, April 29 - May 2, 2019*, pages 325–333, 2019. doi: 10.1109/INFOCOM.2019.8737548. URL https://doi.org/10.1109/INFOCOM.2019.8737548. 99

[139] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan L. Cox. Practical, transparent operating system support for superpages. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*, 2002. URL http://www.usenix.org/events/osdi02/tech/navarro.html. 9, 36, 38, 39, 52, 130, 131

[140] nix Documentation Project. zone - freebsd. https://nixdoc.net/man-pages/FreeBSD/man9/zone.9.html. 36

[141] Oracle. Solaris 11.4 - Locality Groups and Thread and Memory Placement. https://docs.oracle.com/cd/E37838_01/html/E61059/lgroups-32.html, 12 2019. 132

[142] Akash Panda, Ashish Panwar, and Arkaprava Basu. nuksm: Numa-aware memory deduplication on multi-socket servers. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 258–273, 2021. doi: 10.1109/PACT52795.2021.00026. 50

[143] 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 '16, Hong Kong, Hong Kong, 2016. ISBN 9781450342650. doi: 10.1145/2967360.2967371. URL https://doi.org/10.1145/2967360.2967371. 98, 132

[144] 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, pages 679–692, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4911-6. doi: 10.1145/3173162.3173203. URL http://doi.acm.org/10.1145/3173162.3173203. 6, 98

[145] 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, New York, NY, USA, 2019. ACM. 6

[146] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K. Gopinath, and Jayneel Gandhi. *Fast Local Page-Tables for Virtualized NUMA Servers with VMitosis*, page 194–210. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450383172. URL https://doi.org/10.1145/3445814.3446709. 6

[147] M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos. Prediction-based superpage-friendly tlb designs. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 210–222, 2015. doi: 10.1109/HPCA.2015.7056034. 80

[148] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. Page tables: Keeping them flat and hot (cached), 2020. URL https://arxiv.org/abs/2012.05079. 129

[149] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach tlbs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 258–269, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4924-8. doi: 10.1109/MICRO.2012.32. URL https://doi.org/10.1109/MICRO.2012.32. 38, 129

[150] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing tlb reach by exploiting clustering in page translations. *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 558–567, 2014. 129

[151] 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 48th International Symposium on Microarchitecture*, MICRO-48, pages 1–12, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-4034-2. doi: 10.1145/2830772.2830773. URL http://doi.acm.org/10.1145/2830772.2830773. 10, 38

[152] 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 2019, page 333–346, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314582. URL https://doi.org/10.1145/3314221.3314582. 40

[153] 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 '16, pages 99–112, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4091-5. doi: 10.1145/2872362.2872405. URL http://doi.acm.org/10.1145/2872362.2872405. 19, 24

[154] Mitosis Linux Project. Mitosis linux. Online https://github.com/mitosis-project/mitosis-linux/. Accessed 20. May 2020, 2020. 111, 114

[155] Kiran Puttaswamy and Gabriel Loh. Thermal analysis of a 3d die-stacked high-performance microprocessor. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI*, GLSVLSI '06, pages 19–24, New York, NY, USA, 2006. ACM. ISBN 1-59593-347-6. doi: 10.1145/1127908.1127915. URL http://doi.acm.org/10.1145/1127908.1127915. 3, 75

[156] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. Trident: Harnessing architectural resources for all page sizes in x86 processors. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1106–1120, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572. doi: 10.1145/3466752.3480062. URL https://doi.org/10.1145/3466752.3480062. 6

[157] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, page 85–95, Tucson, Arizona, USA, 2011. ISBN 9781450301022. doi: 10.1145/1995896.1995911. URL https://doi.org/10.1145/1995896.1995911. 98

[158] Breno Leitao Rodrigo Ceron, Rafael Folco and Humberto Tsubamoto. Online https://static.rainfocus.com/vmware/vmworldus17/sess/1489512432328001AfWH/

finalpresentationPDF/SER2343BU_FORMATTED_FINAL_1507912874739001gpDS.pdf, 2012. 98

[159] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 469–480, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080210. URL http://doi.acm.org/10.1145/3079856.3080210. 129

[160] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 337–350, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. doi: 10.1145/2465351.2465384. URL http://doi.acm.org/10.1145/2465351.2465384. 131

[161] Dipankar Sarma and Paul E. McKenney. Making rcu safe for deep sub-millisecond response realtime applications. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 32–32, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1247415.1247447. 18

[162] screen.net. Mapping physical memory directly. https://www.sceen.net/mapping-physical-memory-directly/. 36

[163] Amazon Web Services. Amazon ec2 instance types. Online https://aws.amazon.com/ec2/instance-types/, 2020. 99

[164] Riyaj Shamsudeen. Performance tuning: Hugepages in linux. https://blog.pythian.com/performance-tuning-hugepages-in-linux/. 36

[165] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1093–1108, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378493. URL https://doi.org/10.1145/3373376.3378493. 129

[166] Avinash Sodani. Micro keynote: Race to exascale, 2011. https://www.microarch.org/micro44/files/Micro%20Keynote%20Final%20-%20Avinash%20Sodani.pdf. 3, 75

[167] Vijayaraghavan Soundararajan, Mark Heinrich, Ben Verghese, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Flexible use of memory for replication/migration in cache-coherent dsm multiprocessors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, page 342–355, Barcelona, Spain, 1998. ISBN 0818684917. doi: 10.1145/279358.279403. URL https://doi.org/10.1145/279358.279403. 132

[168] Statista. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025. https://www.statista.com/statistics/871513/worldwide-data-created/. x, 2

[169] Guy L. Steele, Jr. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, September 1975. ISSN 0001-0782. doi: 10.1145/361002.361005. URL http://doi.acm.org/10.1145/361002.361005. 131

[170] Darko Stefanović, Kathryn S McKinley, and J Eliot B Moss. Age-based garbage collection. *ACM SIGPLAN Notices*, 34(10):370–381, 1999. 131

[171] Indira Subramanian, Clifford Mather, Kurt Peterson, and Balakrishna Raghunath. Implementation of multiple pagesize support in hp-ux. In *USENIX Annual Technical Conference*, pages 105–119, 1998. 38, 130

[172] Oracle Support. Enable oracle numa support with oracle server. Online https://support.oracle.com/knowledge/Oracle%20Cloud/864633_1.html, 2019. 99

[173] JDK Bug System. Support large pages on macOS. https://bugs.openjdk.java.net/browse/JDK-8233062. 10

[174] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *;login:*, 41(1), 2016. URL https://www.usenix.org/publications/login/spring2016/tarasov. 80

[175] Andrew Theurer. Kvm and big vms. Online https://www.linux-kvm.org/images/5/55/2012-forum-Andrew-Theurer-Big-SMP-VMs.pdf, 2012. 109, 111, 119

[176] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto. 53, 74, 101

[177] Koji Ueno and Toyotaro Suzumura. Highly scalable graph search for the graph500 benchmark. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 149–160, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0805-2. doi: 10.1145/2287076.2287104. URL http://doi.acm.org/10.1145/2287076.2287104. 53, 74, 101

[178] Rik van Riel and Vinod Chegu. Automatic numa balancing. Online https://www.linux-kvm.org/images/7/75/01x07b-NumaAutobalancing.pdf, 2014. 98, 101

[179] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on cc-numa compute servers. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, page 279–289, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917677. doi: 10.1145/237090.237205. URL https://doi.org/10.1145/237090.237205. 132

[180] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 340–349, 2011. doi: 10.1109/PACT.2011.65. 17

[181] vMitosis. vmitosis asplos'21 artifact evaluation. https://github.com/mitosis-project/vmitosis-asplos21-artifact/blob/main/scripts/helpers/fragment.py, 2021. 75

[182] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*, Boston, MA, December 2002. USENIX Association. URL https://www.usenix.org/conference/osdi-02/memory-resource-management-vmware-esx-server. 49, 126, 129

[183] Mitosis workload BTree. Open source code repository. https://github.com/mitosis-project/mitosis-workload-btree, 2019. 101

[184] Zi Yan. 1gb pud thp support on x86_64. https://lwn.net/Articles/832881/, 2020. 131

[185] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, page 331–345, Providence, RI, USA, 2019. ISBN 9781450362405. doi: 10.1145/3297858.3304024. URL https://doi.org/10.1145/3297858.3304024. 97

[186] 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, pages 698–710, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6669-4. doi: 10.1145/3307650.3322223. URL http://doi.acm.org/10.1145/3307650.3322223. 131

[187] Idan Yaniv and Dan Tsafrir. Hash, don't cache (the page table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS '16, page 337–350, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342667. doi: 10.1145/2896377.2901456. URL https://doi.org/10.1145/2896377.2901456. 129

[188] Heechul Yun, Renato Mancuso, Zheng Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, pages 155–166, 2014. doi: 10.1109/RTAS.2014.6925999. URL http://dx.doi.org/10.1109/RTAS.2014.6925999. 37

[189] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 89–102, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-482-9. doi: 10.1145/1519065.1519076. URL http://doi.acm.org/10.1145/1519065.1519076. 53

[190] Weixi Zhu, Alan L. Cox, and Scott Rixner. A comprehensive analysis of superpage management mechanisms and policies. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 829–842. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/zhu-weixi. 130